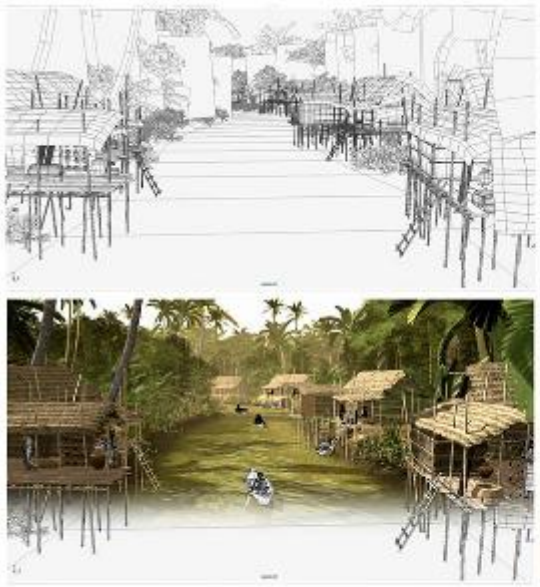
MONASH University

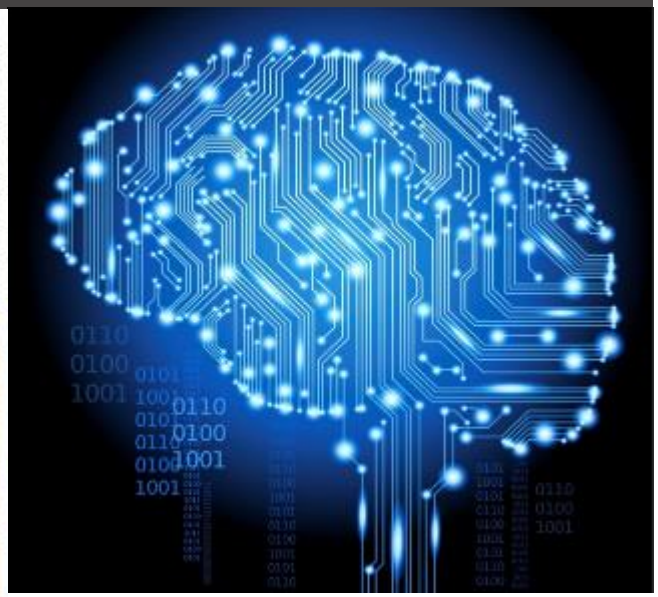
Information Technology

FIT1008/FIT2085 Lecture 9

Function return

Prepared by: M. Garcia de la Banda
based on D. Albrecht, J. Garcia





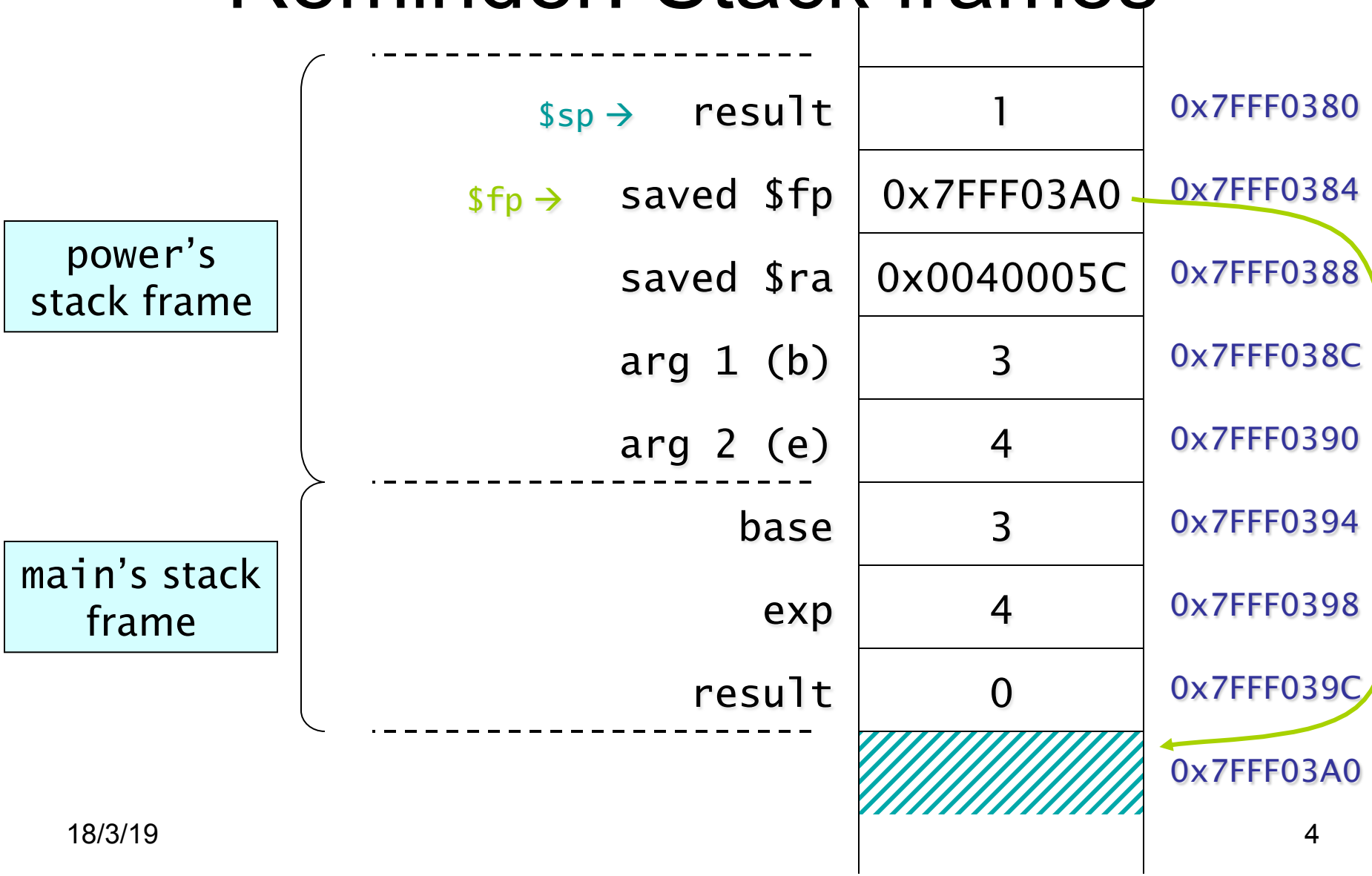
Where are we at:

- **Properties of functions and how they affect their implementation in MIPS**
  - Call and return (`jal` and `jr`)
  - Cascade calls (save **`$ra`** in stack)
  - Access local variables (save **`$fp`** in stack)
  - Pass parameters (use stack)
  - Protect registers (use stack)
- **Calling convention**
  - Who does what and why
  - How a function call is implemented in MIPS
- **Structure of stack**
  - Stack frames

Learning objectives for this lecture

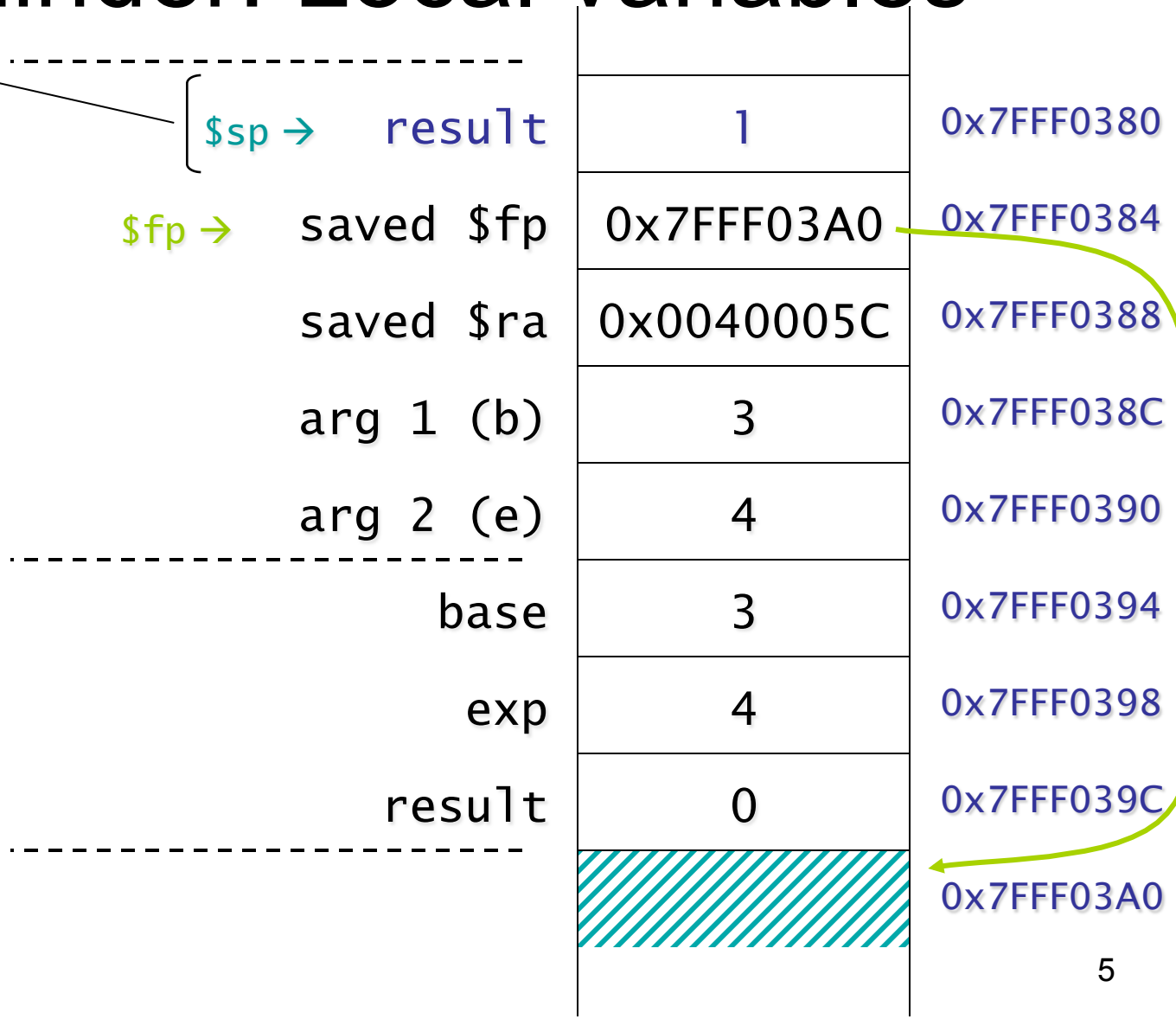
- To understand the steps for function return
- To be able to implement function return in MIPS
- To understand how recursion is implemented in MIPS

Reminder: Stack frames



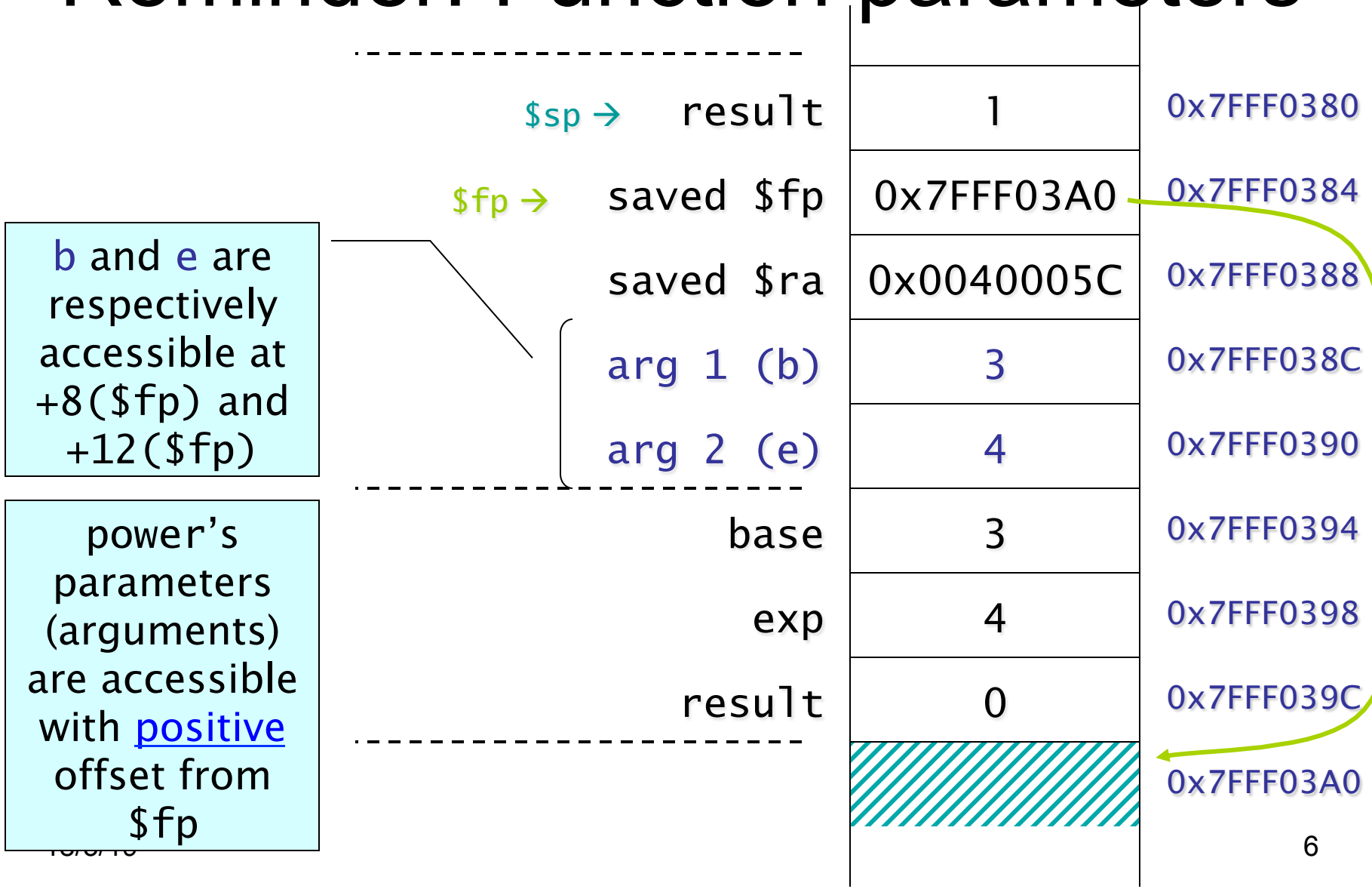
Reminder: Local variables

result is at -4($fp)



power's local variables are accessed as main's were, with negative offsets from $fp

Reminder: Function parameters



Example: callee

```
# A function that gets called.

def power(b, e):

    result = 1

    # Keep going while
    # exponent is positive.

    while e > 0 :

        # Multiply by base.
        result *= b

        # Less to multiply.
        e -= 1

    # Return the result to the caller.
    return result
```

```
# ... Continued from
# last lecture
loop:   # Stop if !(e > 0)
        lw $t0, 12($fp)  # e
        slt $t0, $0, $t0
        beq $t0, 0, end

        # result = result * b
        lw $t0, -4($fp)  # result
        lw $t1, 8($fp)   # b
        mult $t0, $t1
        mflo $t0
        sw $t0, -4($fp)  # result

        # e = e -1
        lw $t0, 12($fp)  # e
        addi $t0, $t0, -1
        sw $t0, 12($fp)  # e

        # Repeat loop.
        j loop

end:    # Now ready to return.
# Continued ...
```

Left it at this point

result is at -4($fp)
b is at 8($fp)
e is at 12($fp)

Function return

- When returning from a function, the stack must be restored to its initial state
- This is achieved by undoing the steps made during calling of function, in reverse order

Function return convention

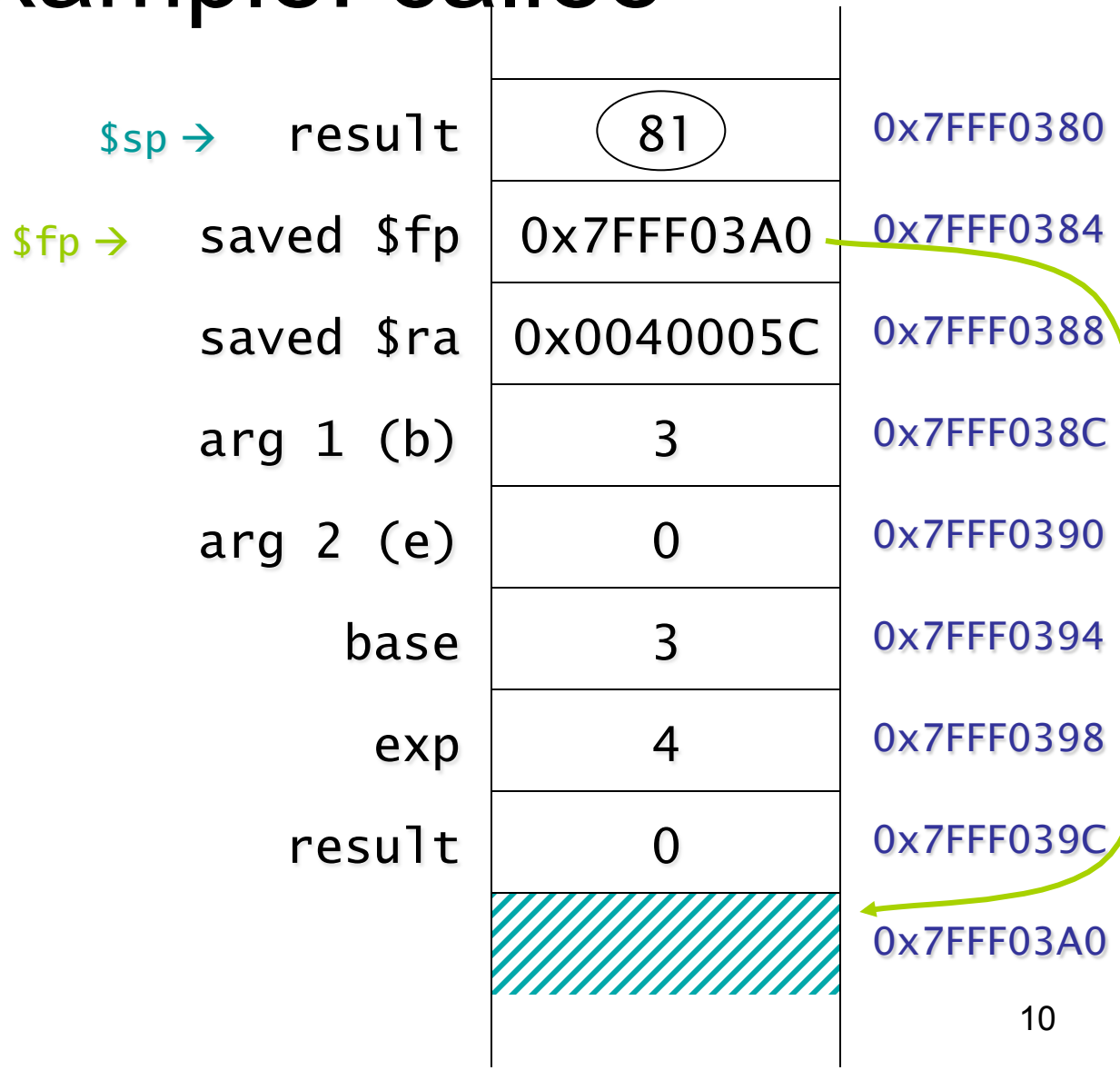
On function exit, callee:

5. Chooses return value by setting register `$v0`, if necessary
6. Deallocates local variables by popping allocated space
7. Restores `$fp` by popping its saved value off stack
8. Restores `$ra` by popping its saved value off stack
9. Returns with `jr $ra`

On return from function, caller:

4. Clears function arguments by popping allocated space
5. Restores saved temporary registers by popping their values off stack
6. Uses the return value found in $v0, if necessary

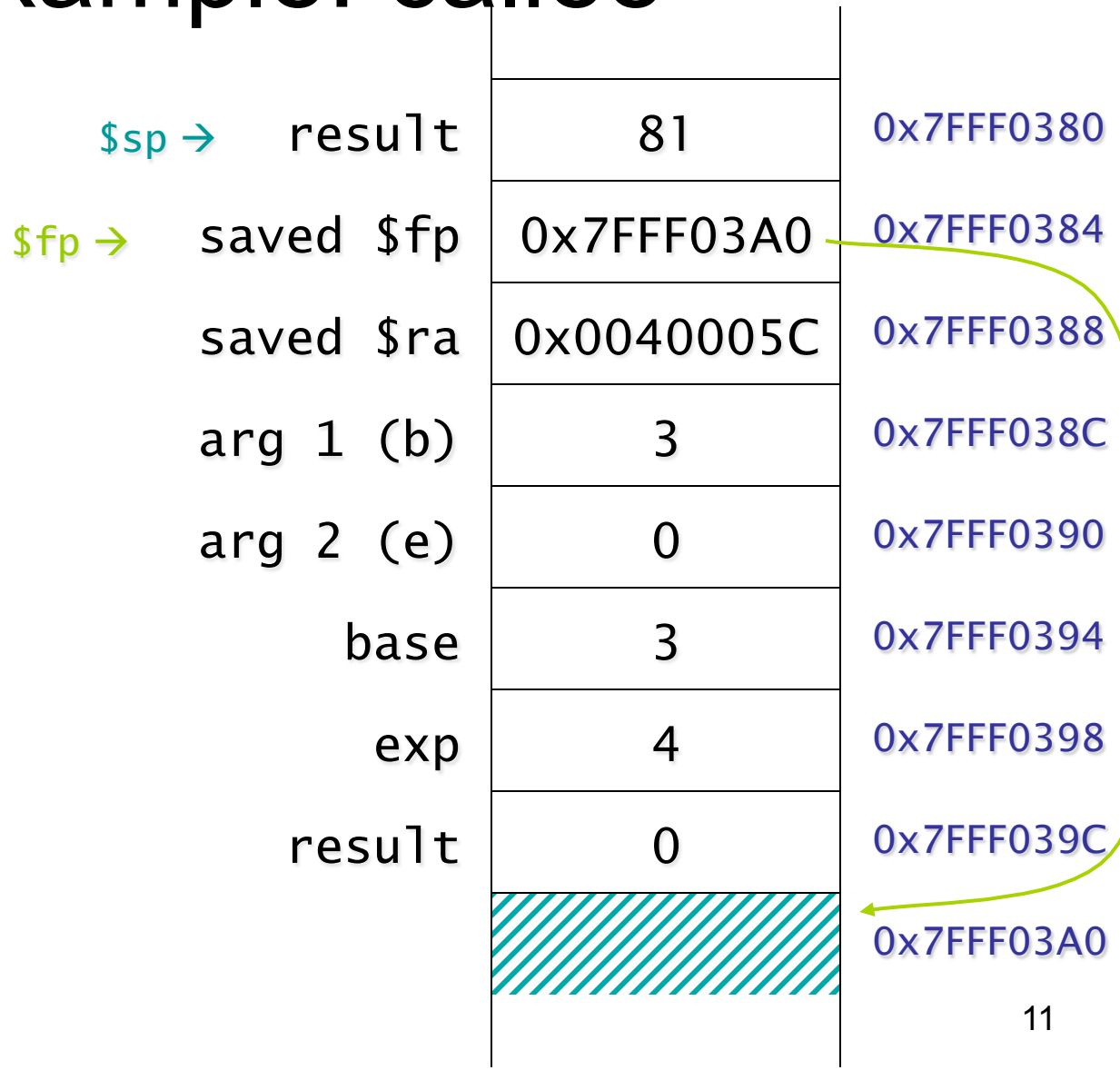
Example: callee



$v0 = 81

callee step 5: put return value in register $v0

Example: callee

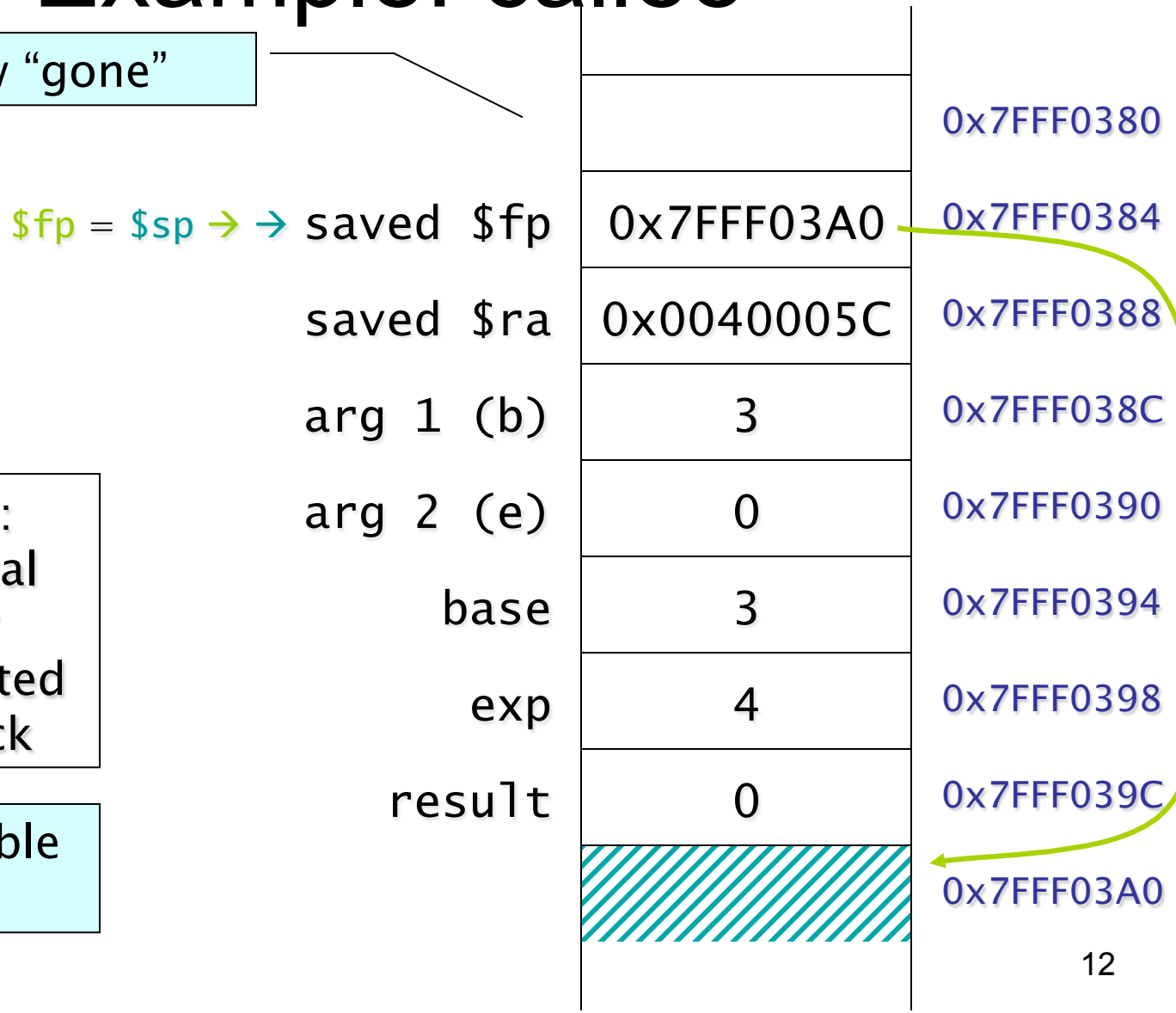


callee step 6: deallocate local variables by popping allocated space off stack

one local variable to delete

Example: callee

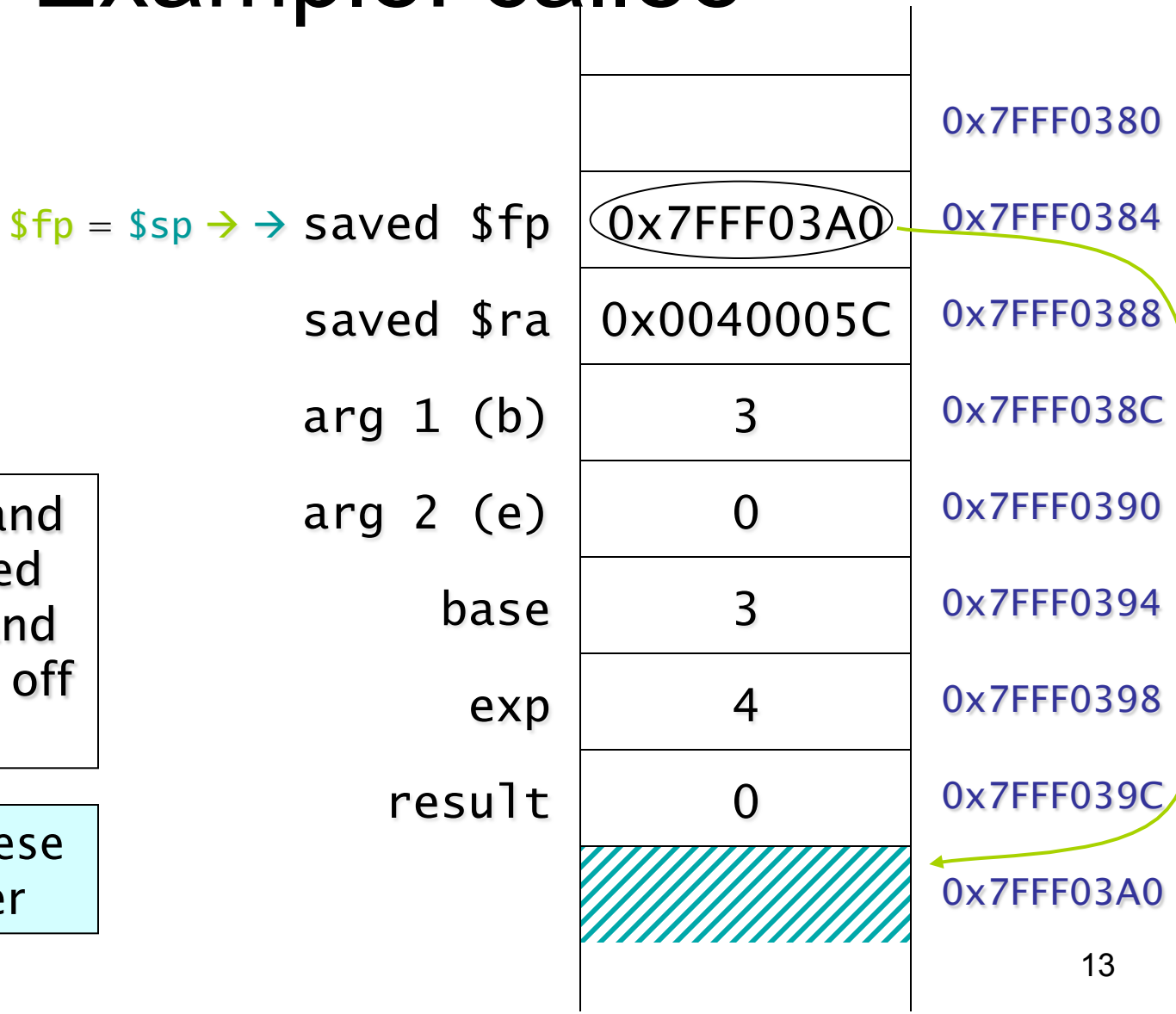
result is now “gone”



callee step 6: deallocate local variables by popping allocated space off stack

one local variable to delete

Example: callee



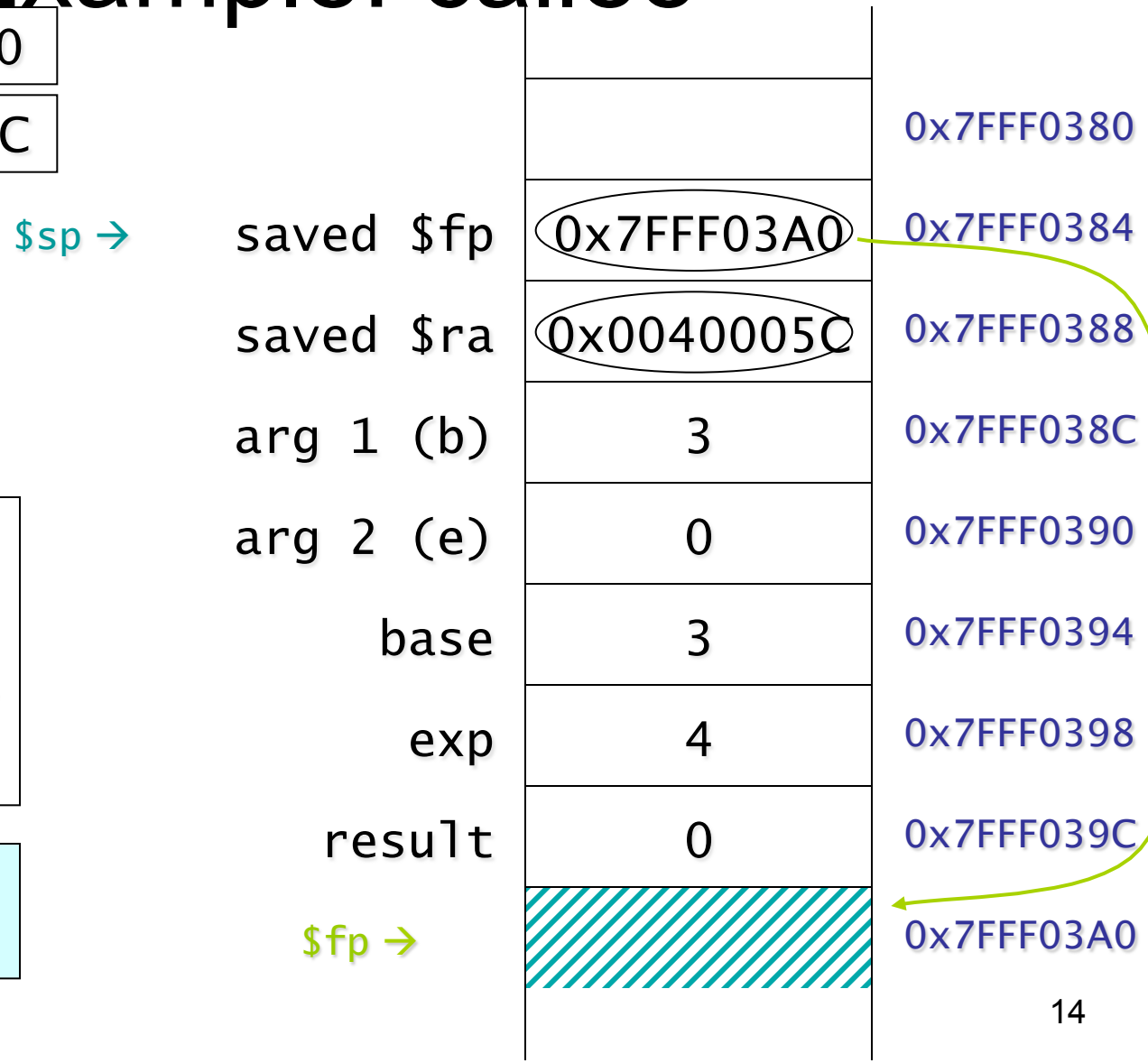
callee steps 7 and 8: restore saved values of $fp and $ra by popping off stack

can do both these steps together

Example: callee

$fp = 0x7FFF03A0

$ra = 0x0040005C



callee steps 7 and 8: restore saved values of $fp and $ra by popping off stack

can do both these steps together

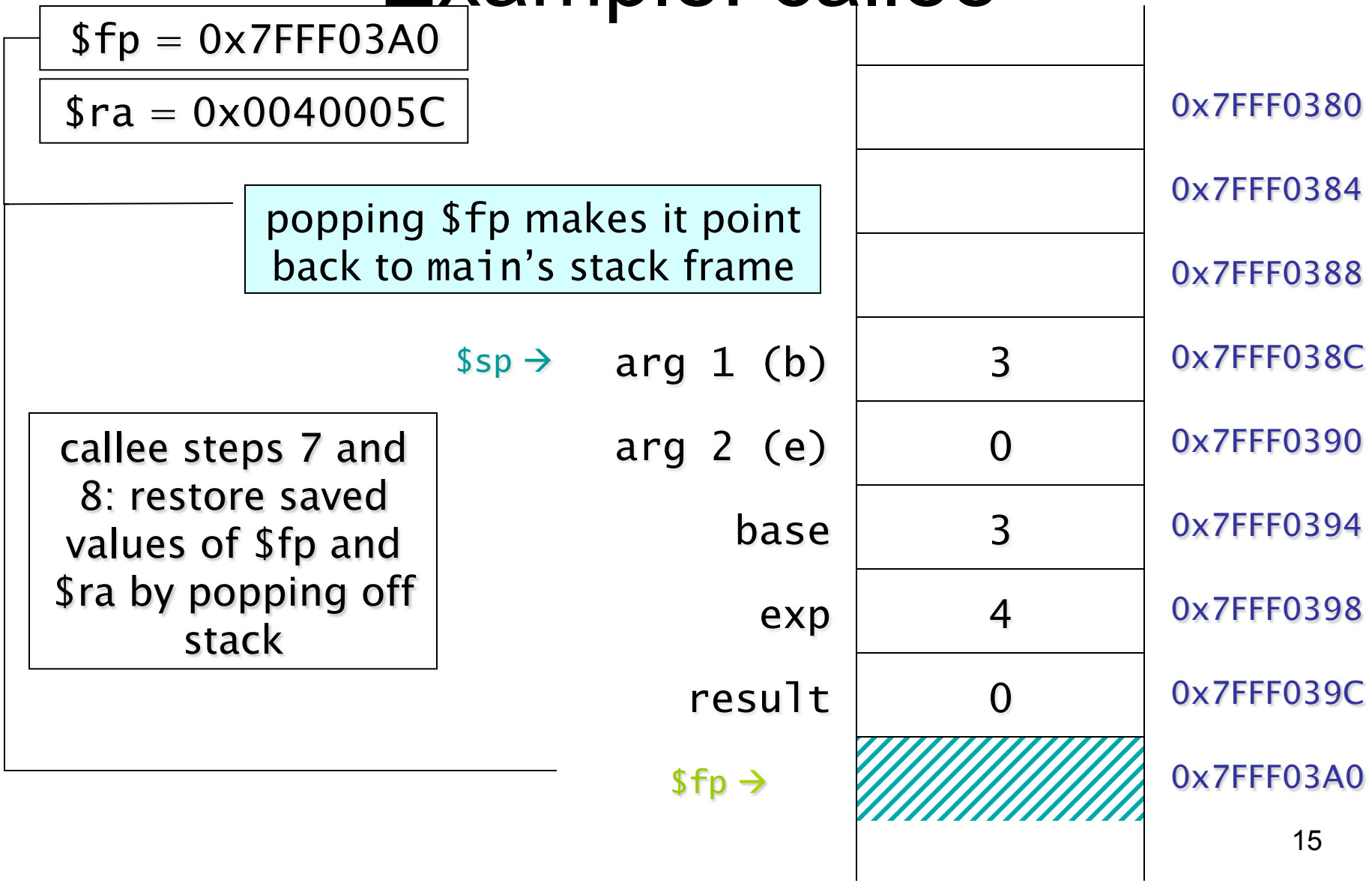
Example: callee

$fp = 0x7FFF03A0

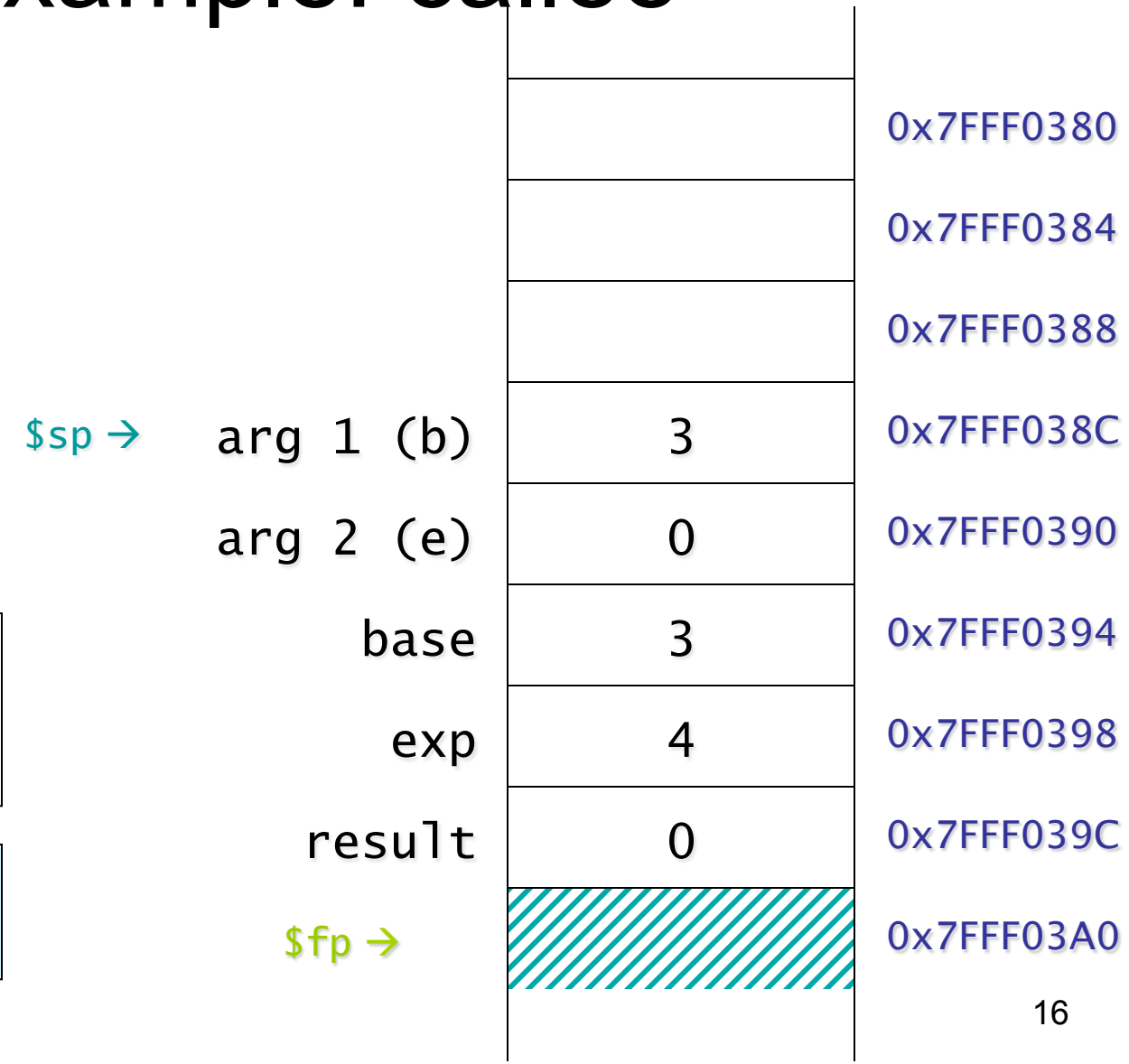
$ra = 0x0040005C

popping $fp makes it point back to main's stack frame

callee steps 7 and 8: restore saved values of $fp and $ra by popping off stack



Example: callee



callee step 9: return by executing `jr $ra`

nothing visible on stack

Example: callee

```
# A function that gets called.

def power(b, e):

    result = 1

    # Keep going while
    # exponent is positive.

    while e > 0 :

        # Multiply by base.
        result *= b

        # Less to multiply.
        e -= 1

    # Return the result to the caller.
    return result
```

```
# ... Continued

# Return result in $v0
lw $v0, -4($fp) # result

# Remove local var.
addi $sp, $sp, 4

# Restore $fp and $ra
lw $fp, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8

# Return to caller.
jr $ra
```

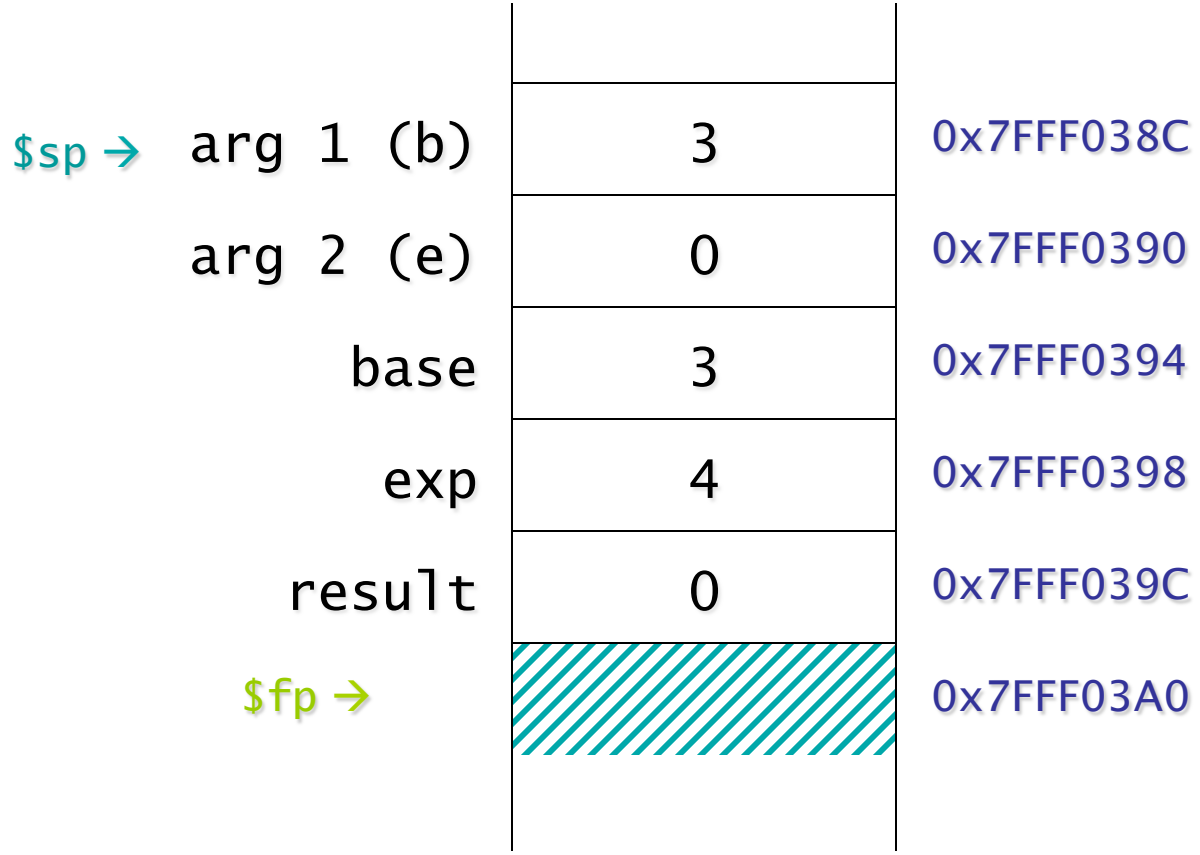
result is at -4($fp)
b is at 8($fp)
e is at 12($fp)

Example: caller

```
# Python program which calls a function.

def power(b, e):
    ...
    return result

def main():
    base, exp, result = 0, 0, 0
    read(base)
    read(exp)

    result = power(base, exp)

    print(result)
```

now back in caller, about to assign return value of function to `main`'s local variable `result`

Example: caller



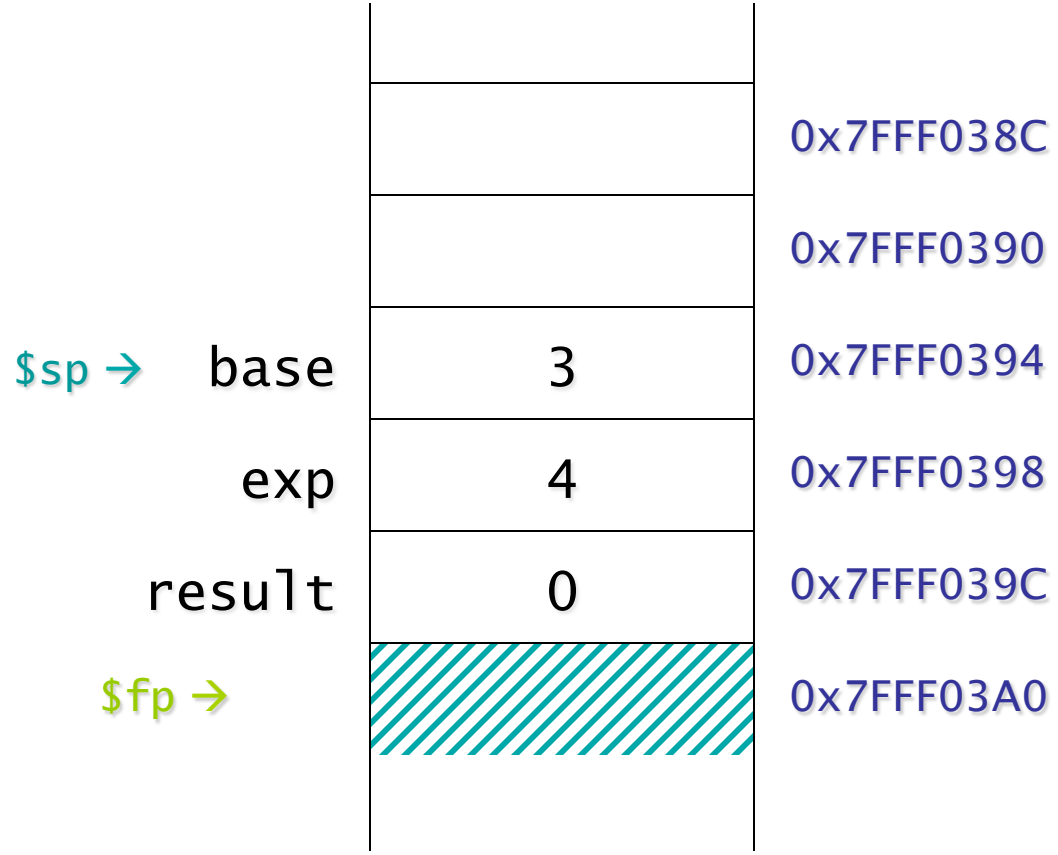
caller step 4: clear function arguments by popping them off stack

b and e are no longer needed, destroy them

Example: caller

caller step 4: clear function arguments by popping them off stack

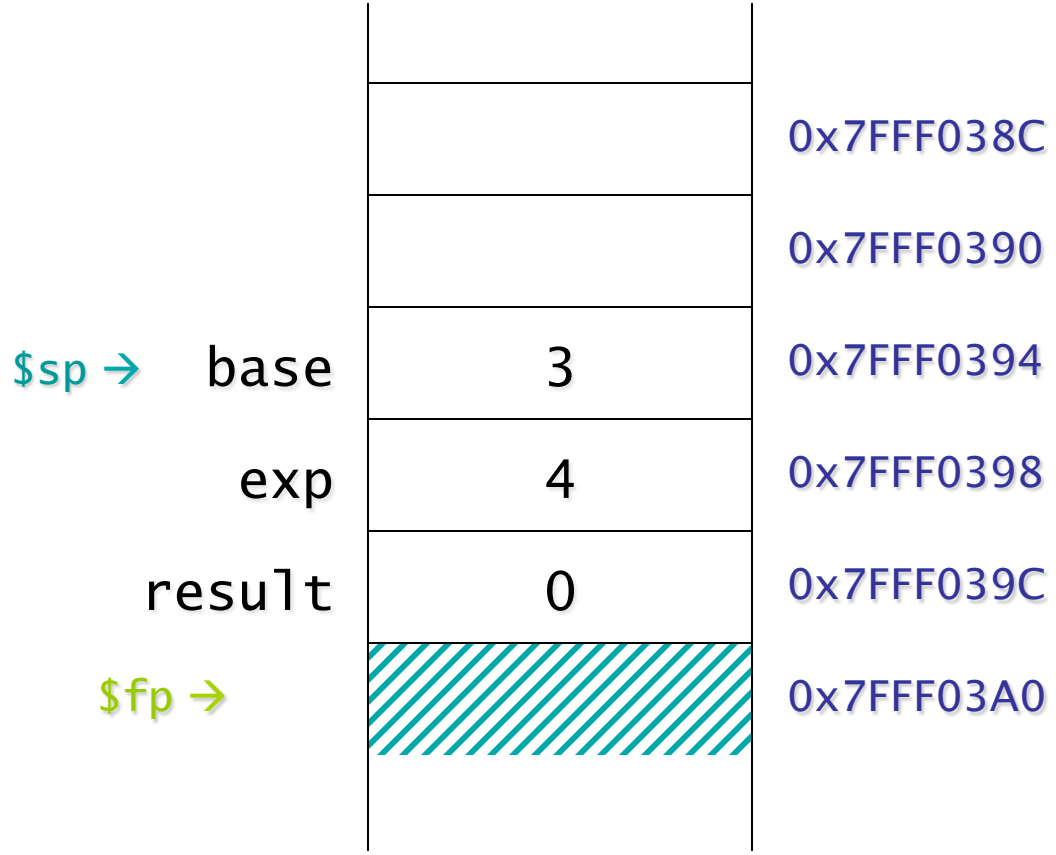
b and e are no longer needed, deallocate them



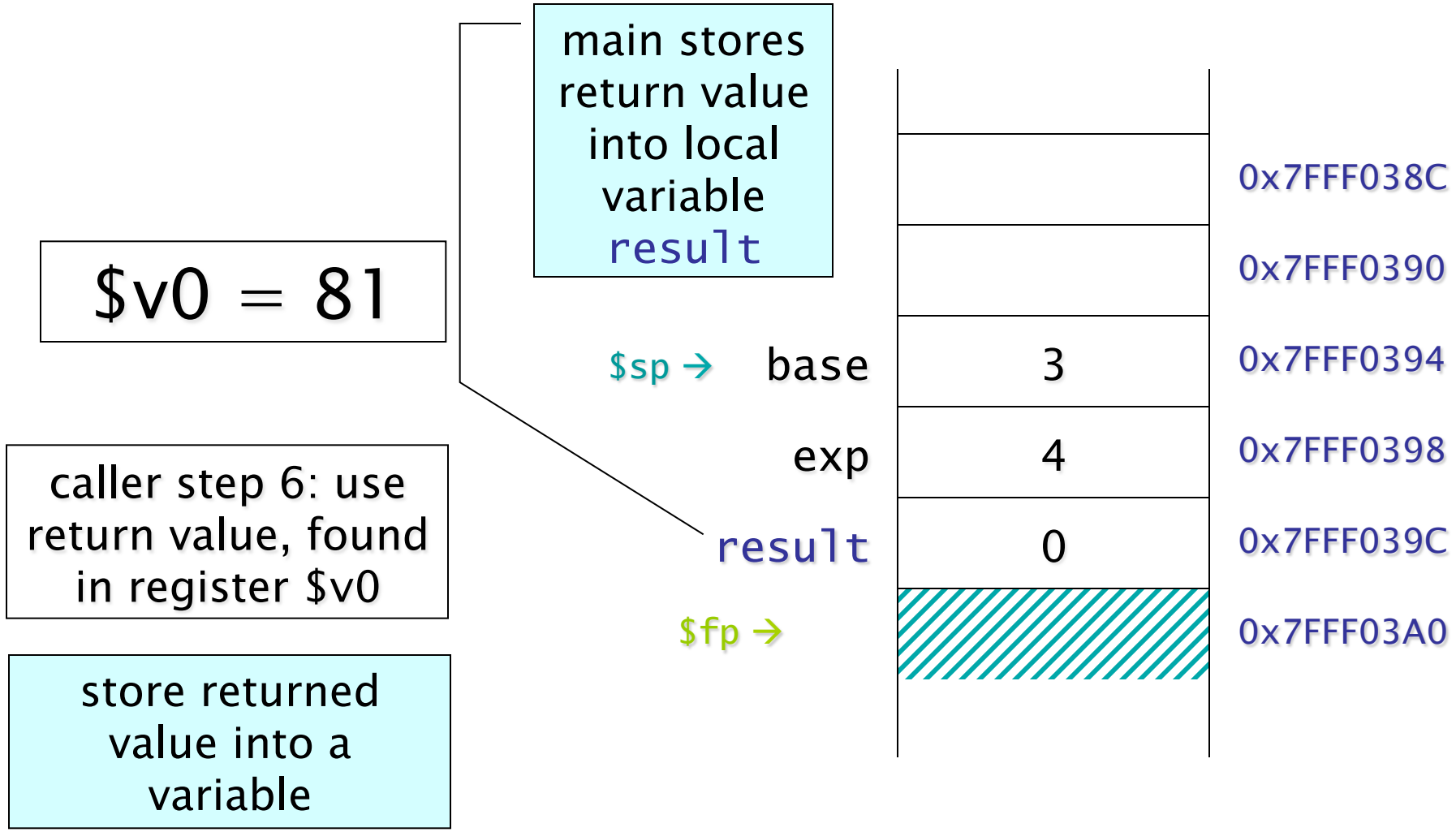
Example: caller

caller step 5: restore saved temporary registers by popping their values off stack

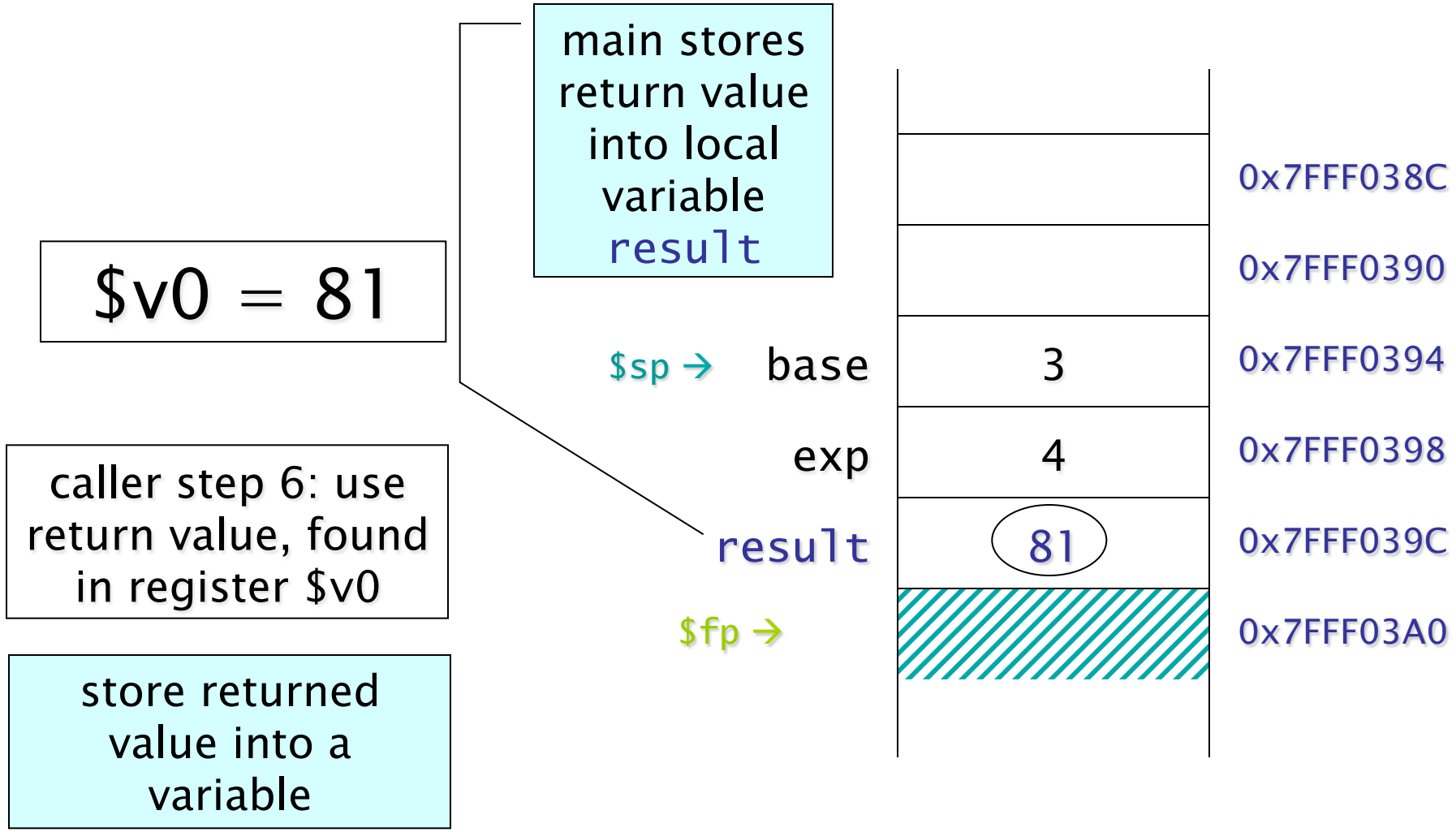
we didn't save any, so nothing to do here



Example: caller



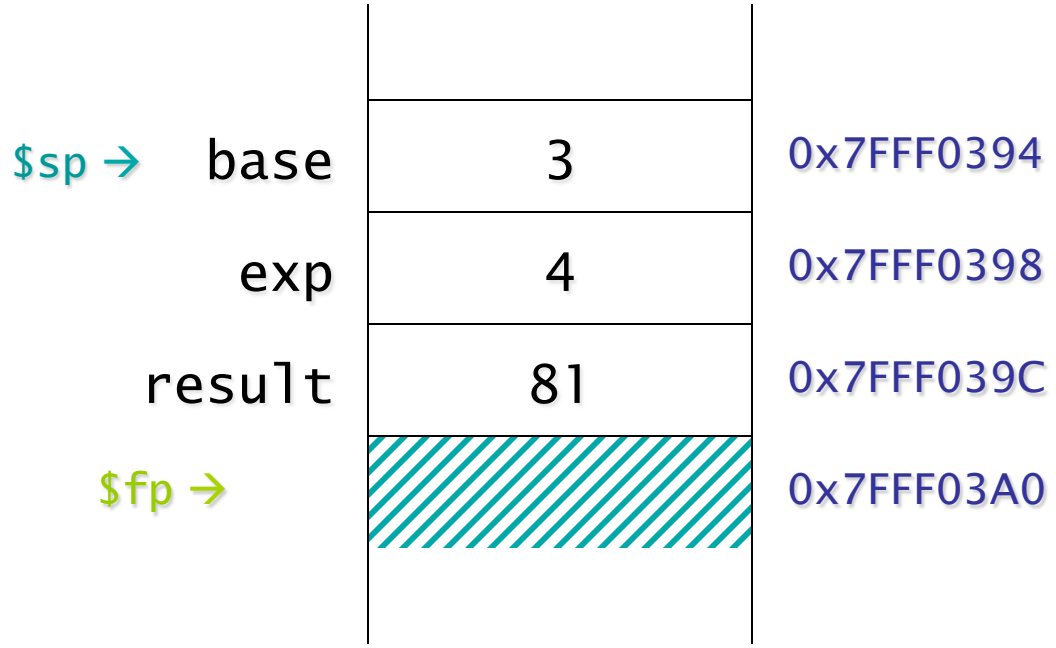
Example: caller



Example: caller

Done!

after the end of the function call the stack has been returned to its original state



Example: caller

```
# Python program which calls a function.

def power(b, e):
    ...
    return result


def main():
    base, exp, result = 0, 0, 0
    read(base)
    read(exp)

    result = power(base, exp)

    print(result)

```

```
# ... main function, continued
        # Last lecture, we
        # finished with this ...
        # jal power

        # Remove arguments, they
        # are no longer needed.
        # 2 * 4 = 8 bytes.
        addi $sp, $sp, 8

        # Store return value
        # in result.
        sw $v0, -4($fp)  # result

        # Print result.
        addi $v0, $0, 1
        lw $a0, -4($fp)  # result
        syscall

        # remove locals, unnecessary
        addi $sp, $sp, 12 #for main
        addi $v0, $0, 10    # exit
        syscall
```

base is at -12($fp)
exp is at -8($fp)
result is at -4($fp)

Function calling convention

In summary, caller:

1. saves temporary registers by pushing their values on stack
2. pushes arguments on stack

3. calls the function with `jal` instruction

– (function runs until it returns, then:)

4. clears function arguments by popping allocated space
5. restores saved temporary registers by popping their values off the stack
6. uses the return value found in $v0

Function calling convention

In summary, callee:

1. saves `$ra` by pushing its value on stack
2. saves `$fp` by pushing its value on stack
3. copies `$sp` to `$fp`
4. allocates local variables

– (body of function goes here, then:)

5. chooses return value by setting register `$v0`
6. deallocates local variables by popping allocated space
7. restores `$fp` by popping its saved value
8. restores `$ra` by popping its saved value
9. returns with `jr $ra`

Recursion

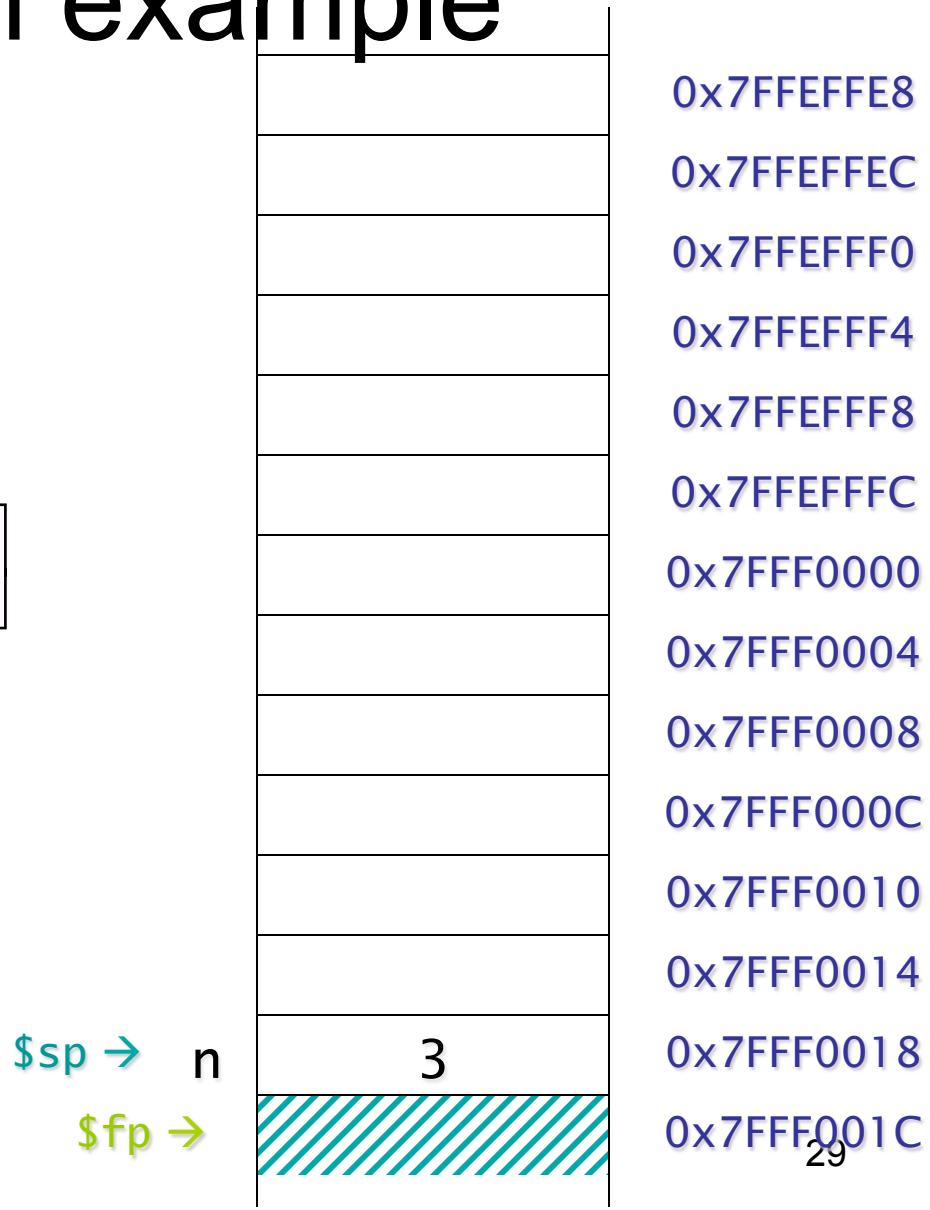
- **Recursive functions end up calling themselves**
  - Solving a simpler/smaller version of the same problem
- **Function calling convention works exactly the same for recursive functions**
  - Don't need to do anything special!!
- **Each invocation of the function has its own stack frame**
  - Local variables and parameters
    - With their current values
  - Return address
    - Where to return to

```
# The factorial function.

def factorial(p):
    result = 0
    if p <= 1 :  # Base case
        result = 1
    else:        # Recursive case
        result =
            factorial(p - 1) * p
    return result
```

Recursion example

```
# Main program to call
# factorial function.

def factorial(p):
    ...
    return result

def main():
    n = 0

    read(n)

    print(factorial(n))
```

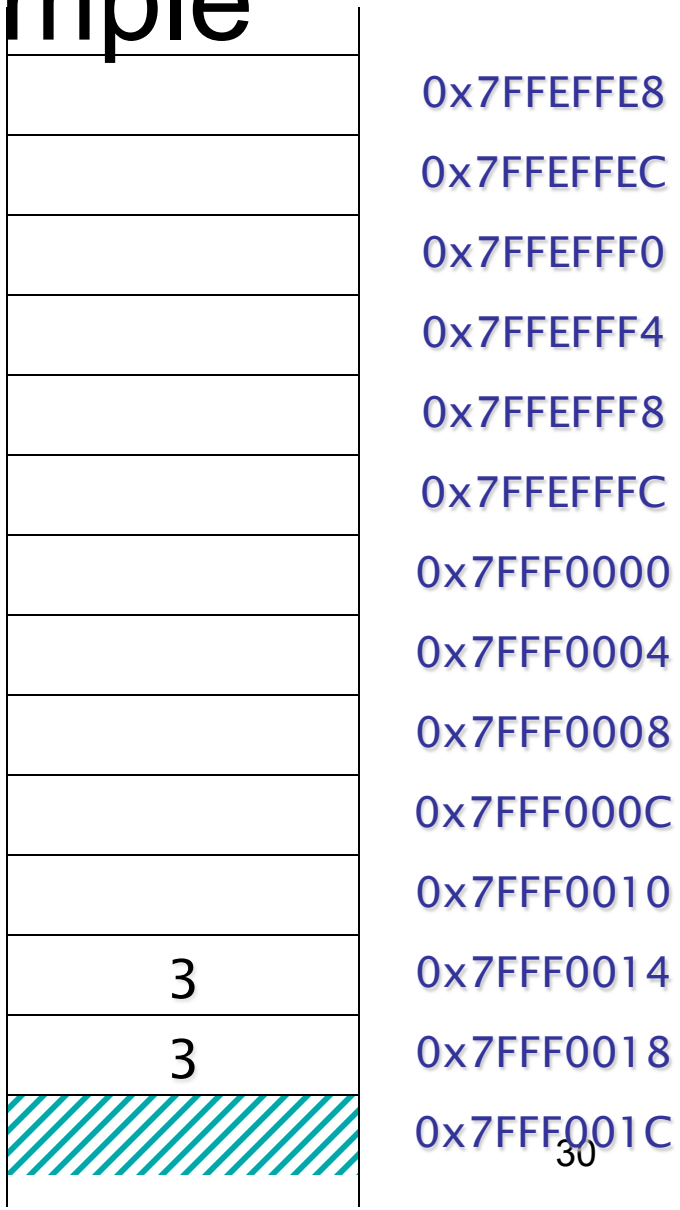
Assume you read 3 into n



Recursion example

```
# Main program to call
# factorial function.

def factorial(p):
    ...
    return result

def main():
    n = 0

    read(n)

    print(factorial(n))
```

Assume you read 3 into n



From the point of view of main, the argument is at 0($sp)

Recursion example: caller

```
# Main program to call
# factorial function.

def factorial(p):
    ...
    return result

def main():
    n = 0

    read(n)

    print(factorial(n))
```

```
        .text
main:   # 1 * 4 = 4 bytes local.
        addi $fp, $sp, 0
        addi $sp, $sp, -4
        sw   $0, -4($fp) # n = 0

        addi $v0, $0, 5
        syscall
        sw $v0, -4($fp)  # n

        # Call factorial.
        # 1 * 4 = 4 bytes arg.
        addi $sp, $sp, -4
        lw $t0, -4($fp)  # n
        sw $t0, 0($sp)   # arg 1
        jal factorial
        # Clear argument.
        addi $sp, $sp, 4
        # Print result
        addi $a0, $v0, 0 # returned
        addi $v0, $0, 1  # print int
        syscall

        # remove locals, unnecessary
        addi $sp, $sp, 4 # for main
        addi $v0, $0, 10 # exit
        syscall
```

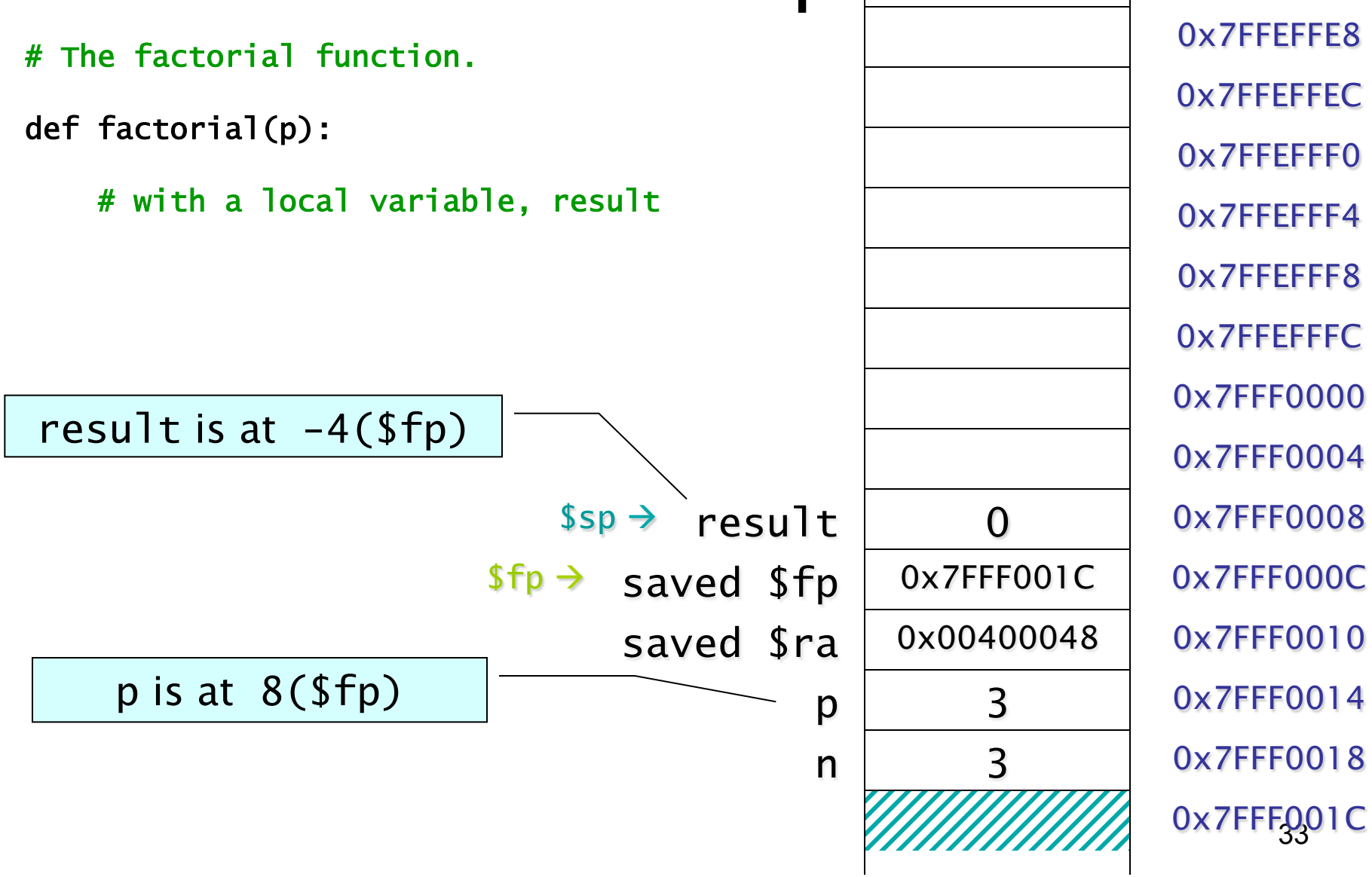
n is at -4($fp)

Recursion example: callee

```
# The factorial function.

def factorial(p):
    result = 0

    if p <= 1 :
        # Base case
        result = 1

    else:
        # Recursive case
        result =
            factorial(p - 1) * p

    return result
```

Recursion example: callee

```
# The factorial function.

def factorial(p):

    # with a local variable, result
```



Recursion example: callee

```
# The factorial function.

def factorial(p):

    result = 0

    if p <= 1 :
        # Base case
        result = 1

    else:
        # Recursive case
        result =
            factorial(p-1) * p;

    return result
```

```
factorial:   # Function entry
        addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $fp, 0($sp)
        addi $fp, $sp, 0

        # 1 * 4 = 4 bytes local
        addi $sp, $sp, -4
        sw   $0, -4($fp) # result = 0

        # if p <= 1 ...
        lw $t0, 8($fp)  # p
        addi $t1, $0, 1
        slt $t0, $t1, $t0
        bne $t0, $0, rec

        # result = 1
        addi $t0, $0, 1
        sw $t0, -4($fp)  # result
        j end

# Continued ...
```

p is at 8($fp)
result is at -4($fp)

Recursion example: callee

```
# The factorial function.

def factorial(p):

    result = 0

    if p <= 1 :
        # Base case
        result = 1

    else:
        # Recursive case
        result =
            factorial(p-1) * p

    return result
```

```
# ... Continued
rec:    # Recursive call.
        # 1 * 4 = 4 bytes arg.
        addi $sp, $sp, -4

        # argument 1 = p-1
        lw $t0, 8($fp)     # p
        addi $t0, $t0, -1 # p-1
        sw $t0, 0($sp)     # arg 1

        jal factorial

        # Clean up argument.
        addi $sp, $sp, 4

        # Multiply by p.
        lw $t0, 8($fp)  # p
        mult $v0, $t0
        mflo $t0

        # Store result.
        sw $t0, -4($fp)  # result

# Continued ...
```

p is at 8($fp)
result is at -4($fp)

Recursion example: callee

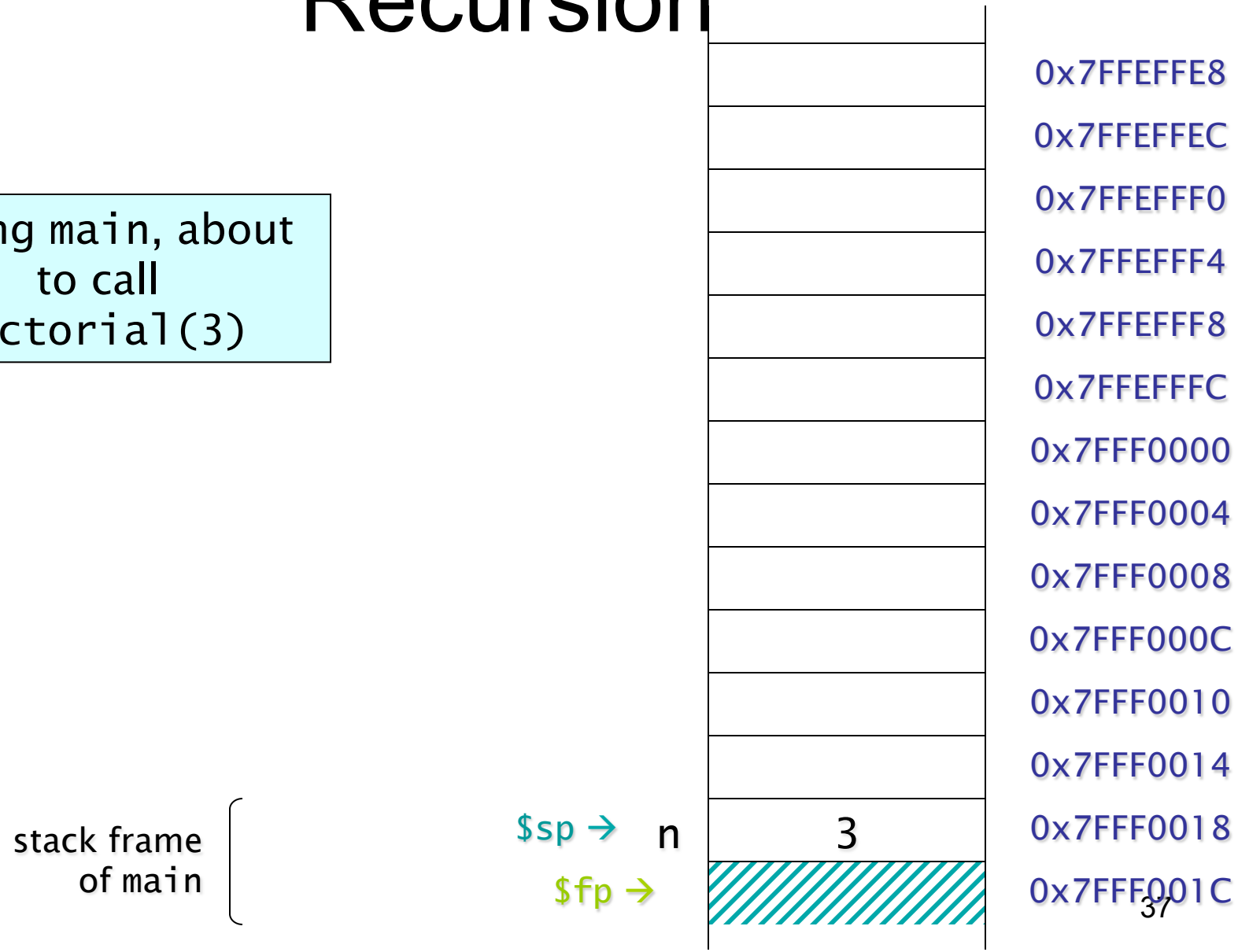
```
# The factorial function.

def factorial(p):

    result = 0

    if p <= 1 :
        # Base case
        result = 1

    else:
        # Recursive case
        result =
            factorial(p-1) * p

    return result
```

```
# ... Continued again

end:    # return result
        lw $v0, -4($fp)  # result

        # Destroy local variable
        addi $sp, $sp, 4

        # Function exit.
        lw $fp, 0($sp)
        lw $ra, 4($sp)
        addi $sp, $sp, 8
        jr $ra
```

p is at 8($fp)
result is at -4($fp)

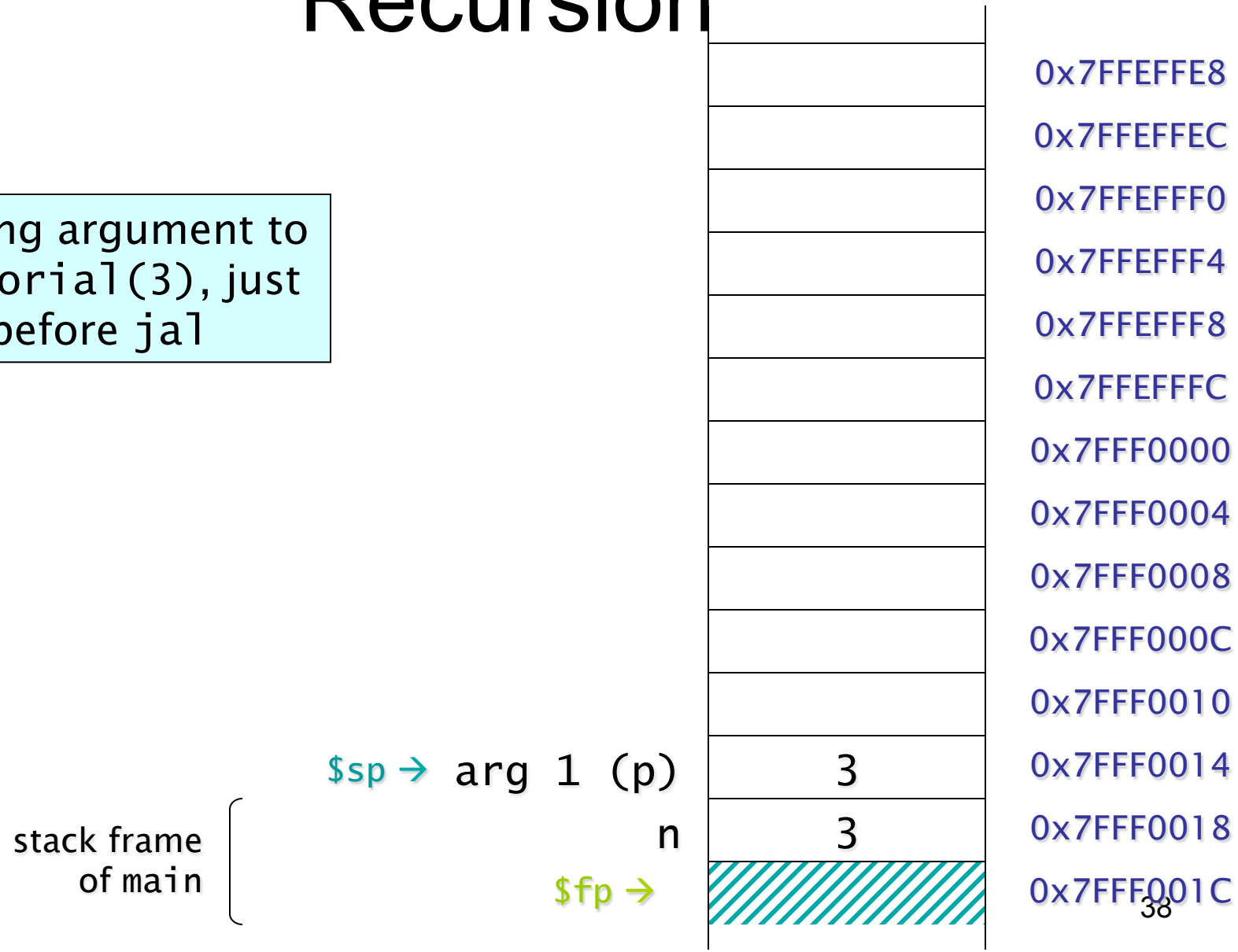
Recursion

during `main`, about to call `factorial(3)`



Recursion

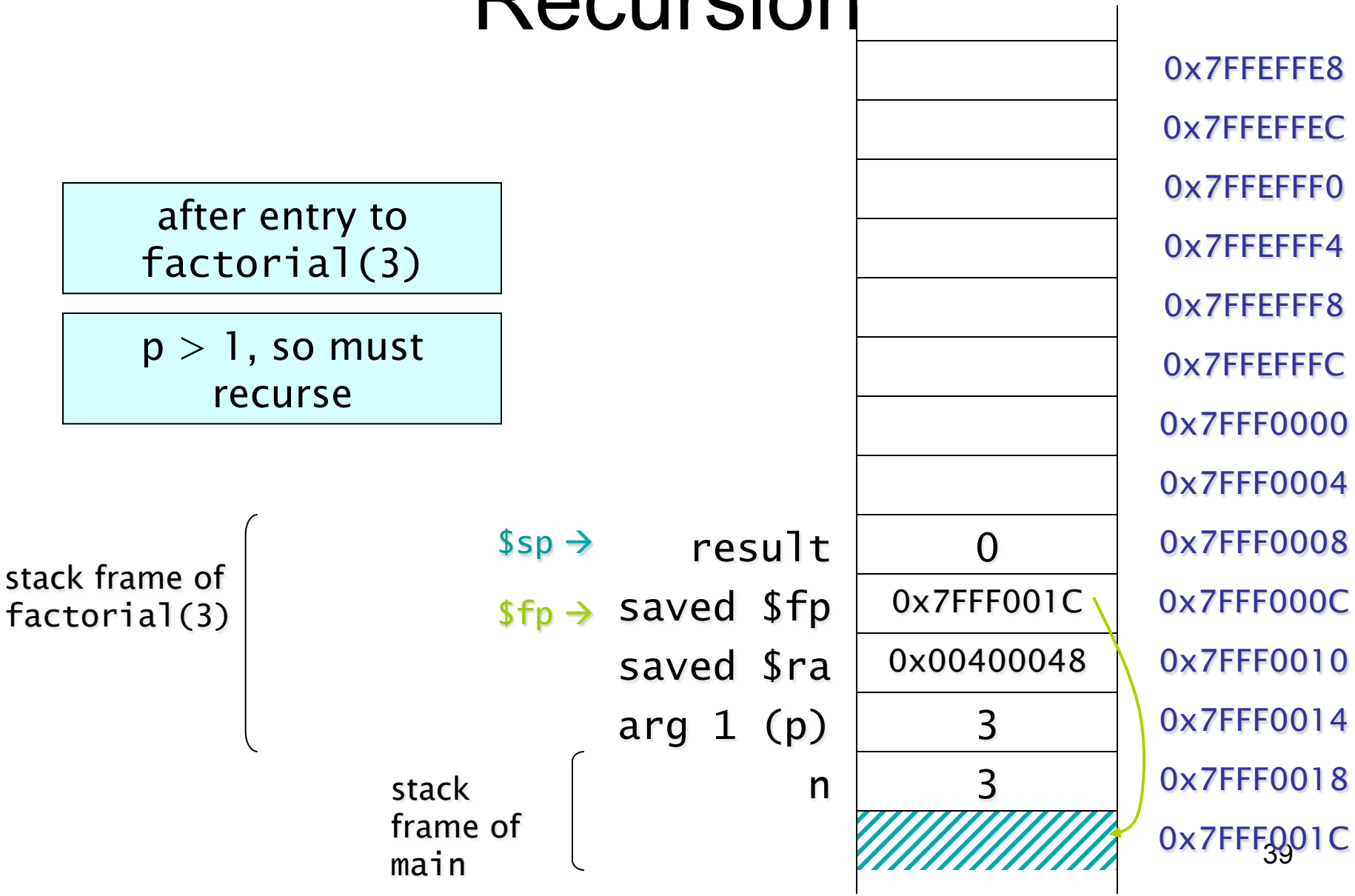
passing argument to `factorial(3)`, just before `jal`



Recursion

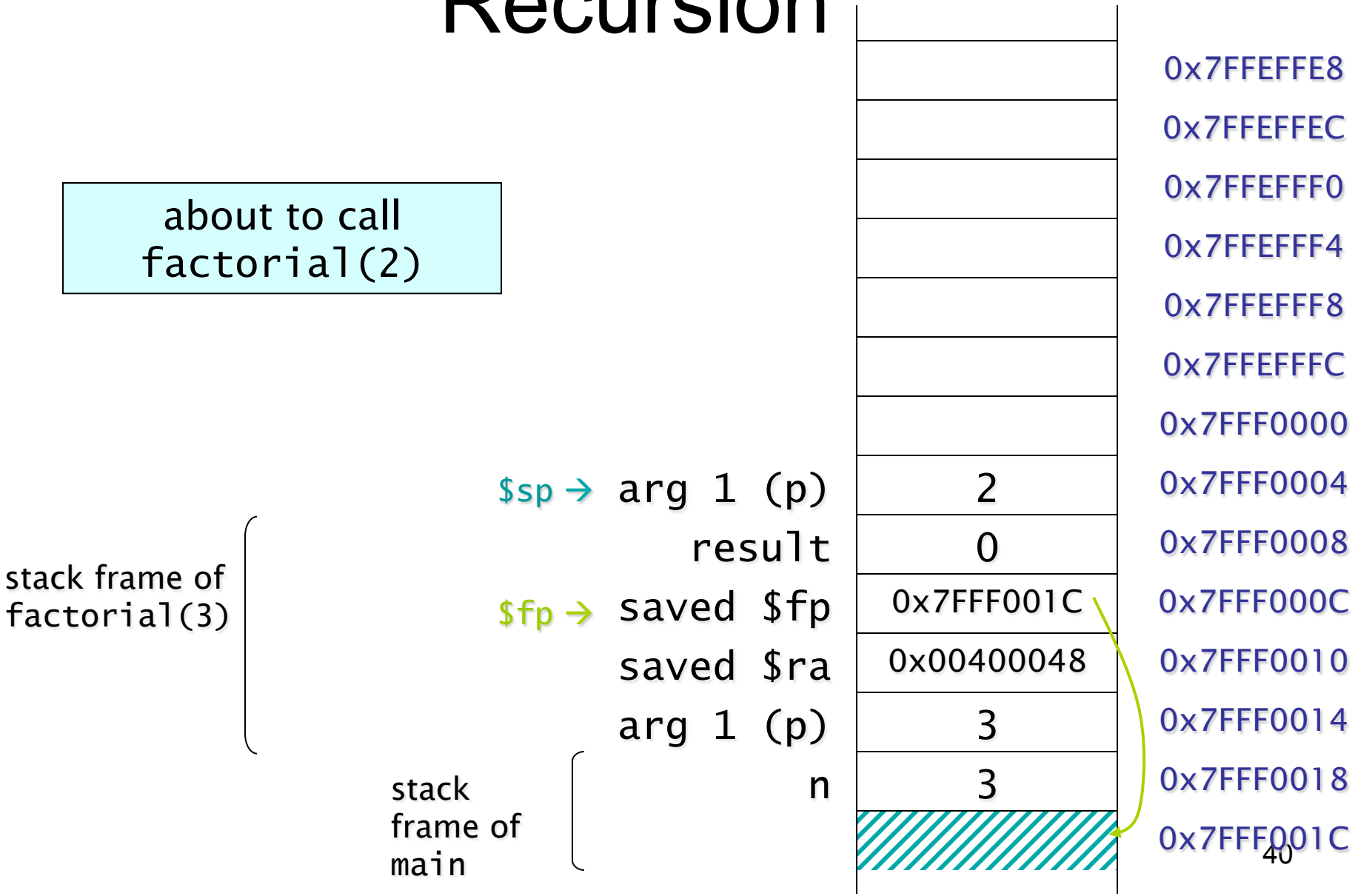
after entry to `factorial(3)`

`p > 1`, so must recurse



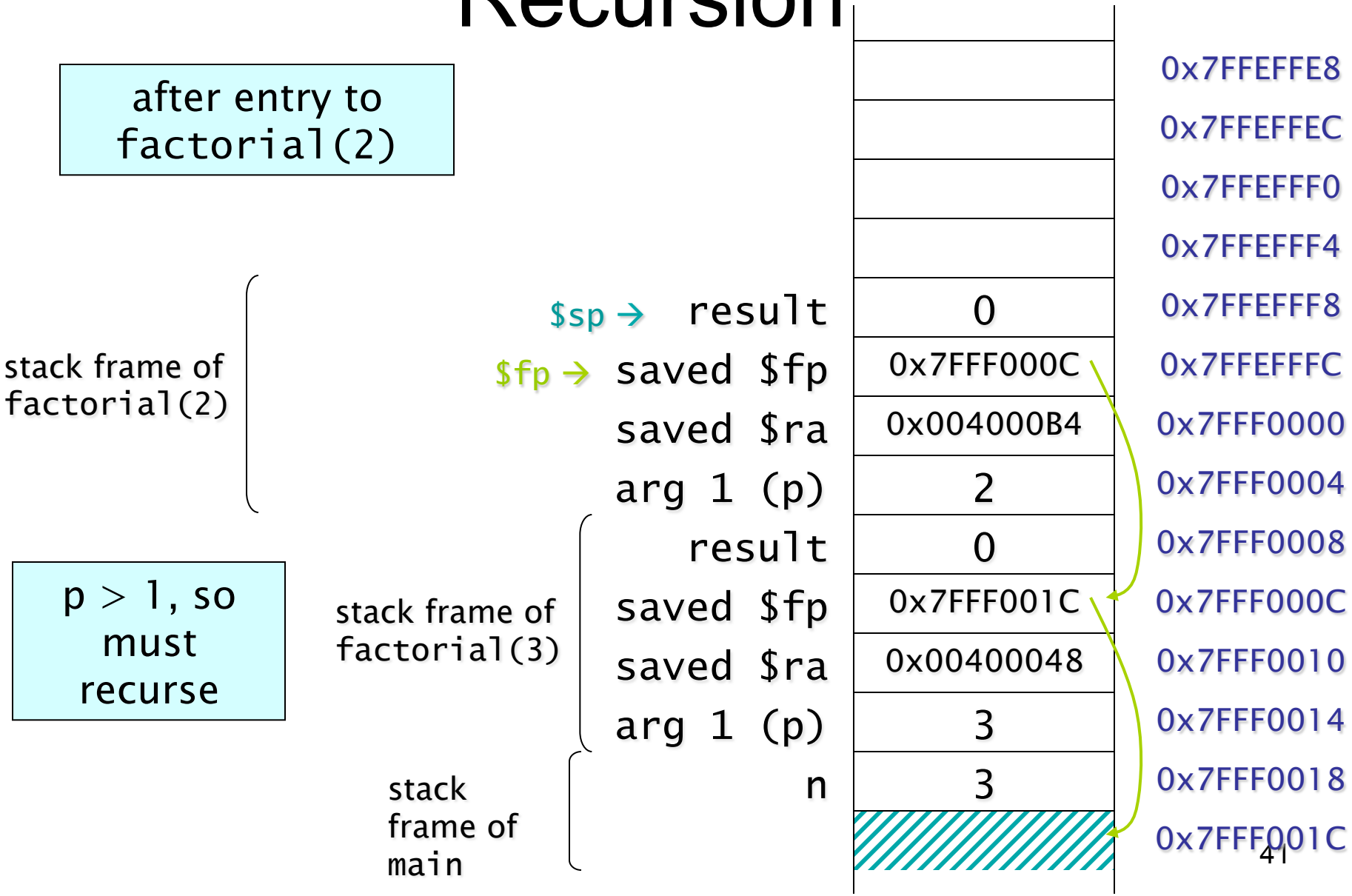
Recursion

about to call factorial(2)



Recursion

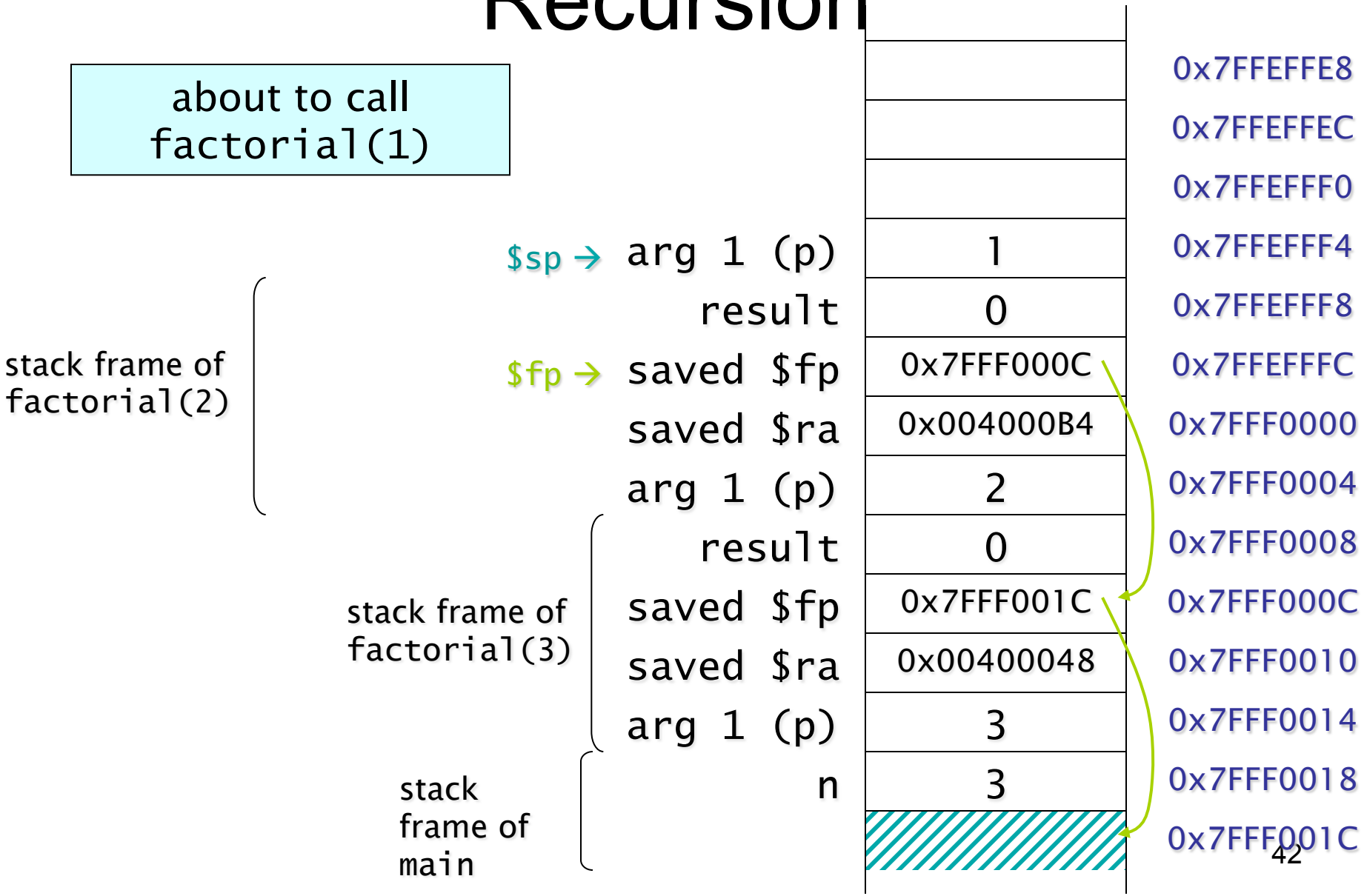
after entry to factorial(2)



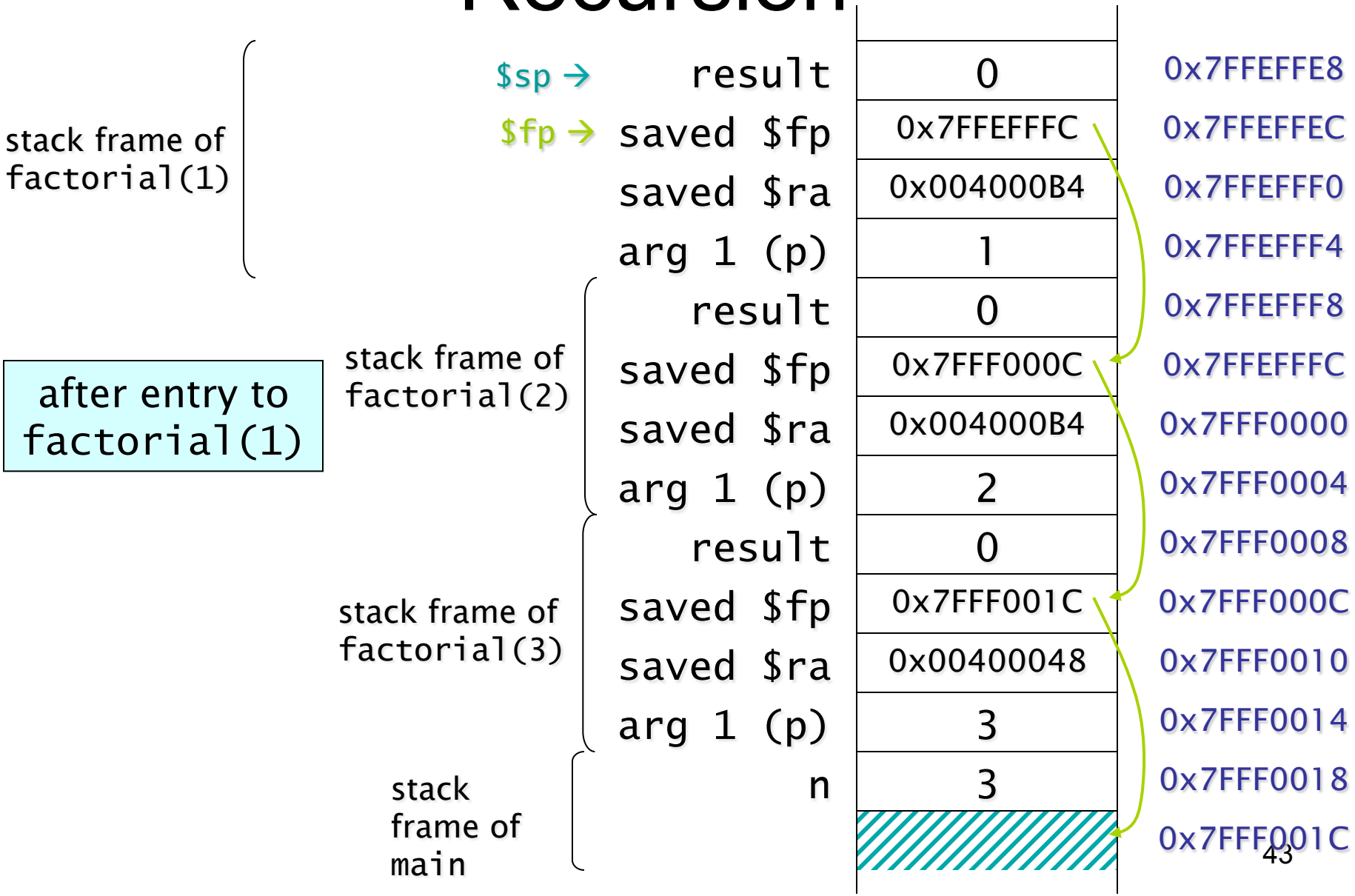
p > 1, so must recurse

Recursion

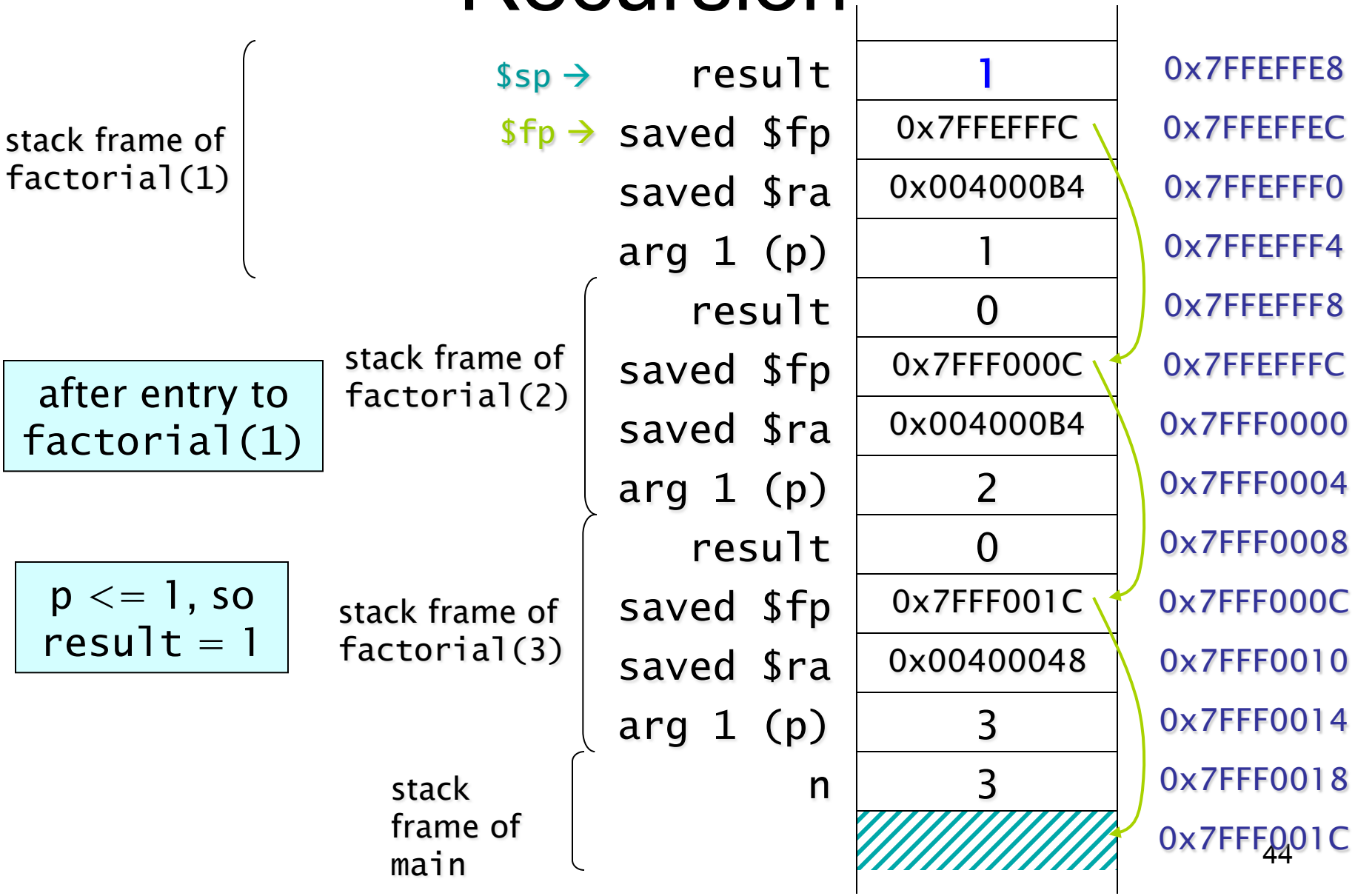
about to call factorial(1)



Recursion

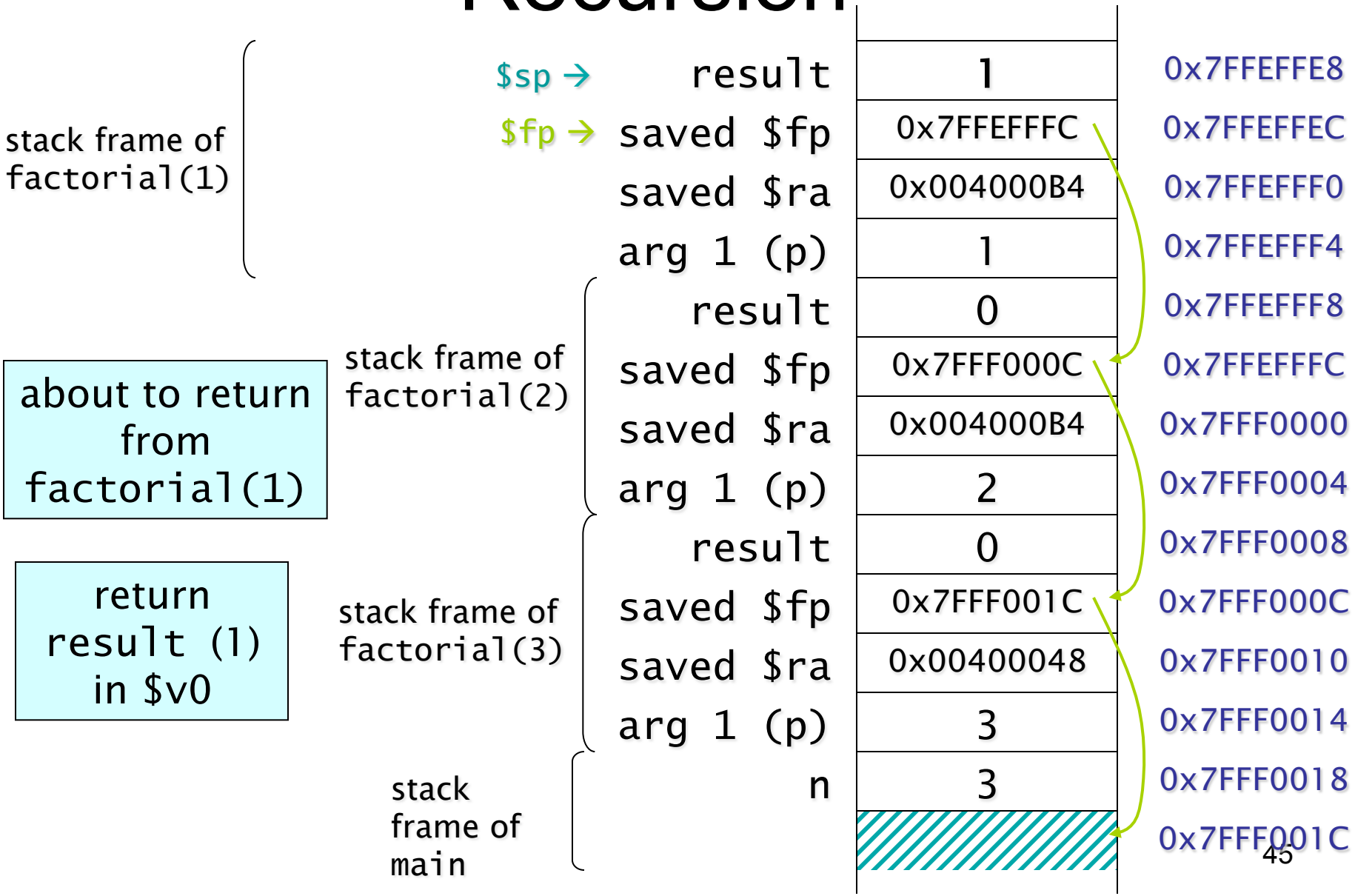

stack frame of factorial(1)
$sp →
$fp →
result
saved $fp
saved $ra
arg 1 (p)
0
0x7FFEFFFC
0x004000B4
1
0x7FFEFFE8
0x7FFEFFEC
0x7FFEFFF0
0x7FFEFFF4
after entry to factorial(1)
stack frame of factorial(2)
result
saved $fp
saved $ra
arg 1 (p)
0
0x7FFF000C
0x004000B4
2
0x7FFEFFF8
0x7FFEFFFC
0x7FFF0000
0x7FFF0004
stack frame of factorial(3)
result
saved $fp
saved $ra
arg 1 (p)
0
0x7FFF001C
0x00400048
3
0x7FFF0008
0x7FFF000C
0x7FFF0010
0x7FFF0014
stack frame of main
n
3
0x7FFF0018
0x7FFF001C

Recursion


stack frame of factorial(1)
after entry to factorial(1)
p <= 1, so result = 1
stack frame of factorial(2)
stack frame of factorial(3)
stack frame of main
$sp →
$fp →
result
saved $fp
saved $ra
arg 1 (p)
result
saved $fp
saved $ra
arg 1 (p)
result
saved $fp
saved $ra
arg 1 (p)
n
1
0x7FFEFFFC
0x004000B4
1
0
0x7FFF000C
0x004000B4
2
0
0x7FFF001C
0x00400048
3
3
0x7FFEFFE8
0x7FFEFFEC
0x7FFEFFF0
0x7FFEFFF4
0x7FFEFFF8
0x7FFEFFFC
0x7FFF0000
0x7FFF0004
0x7FFF0008
0x7FFF000C
0x7FFF0010
0x7FFF0014
0x7FFF0018
0x7FFF001C

Recursion

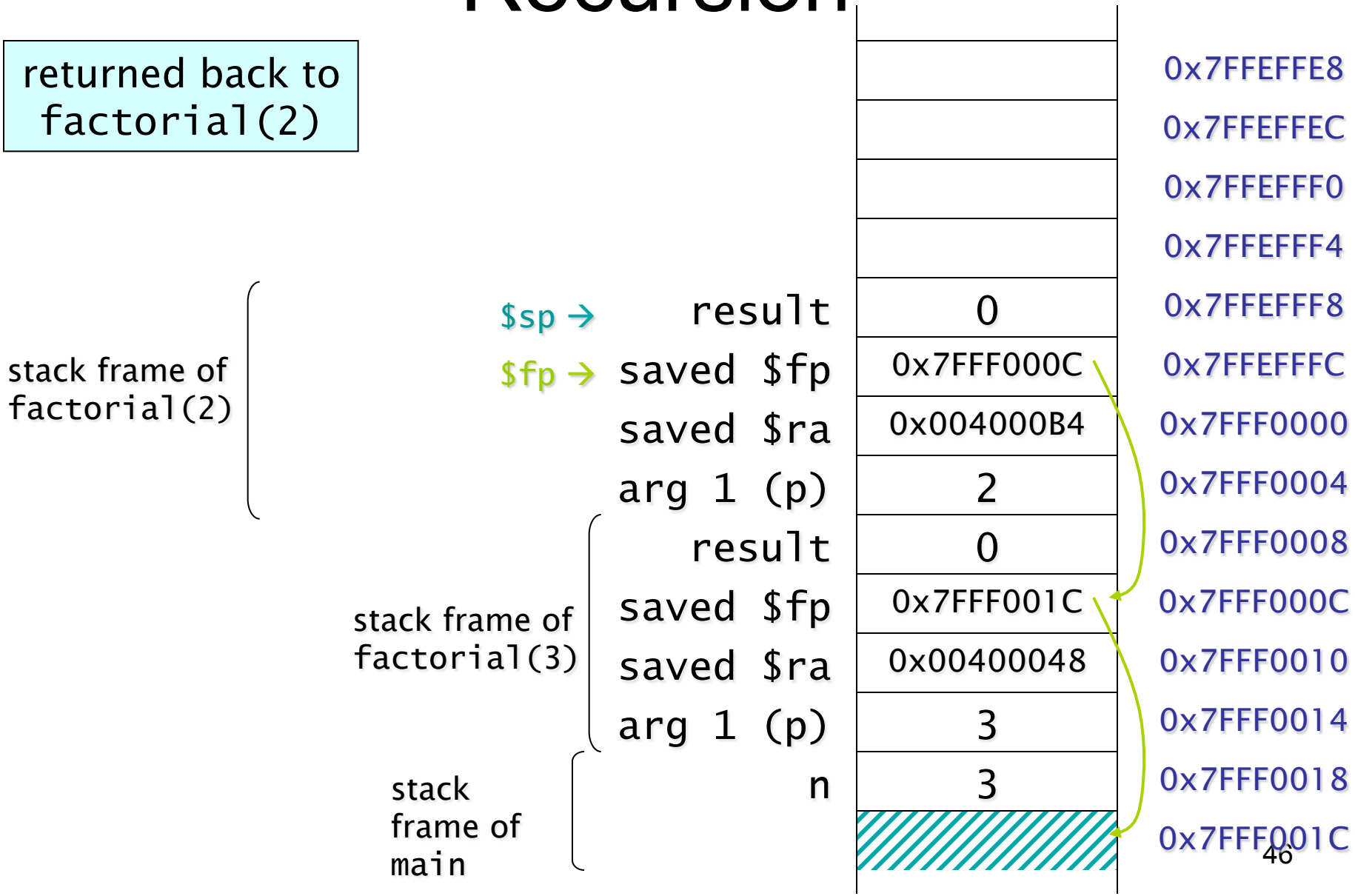
$v0 = 1



$v0 = 1

Recursion

returned back to factorial(2)

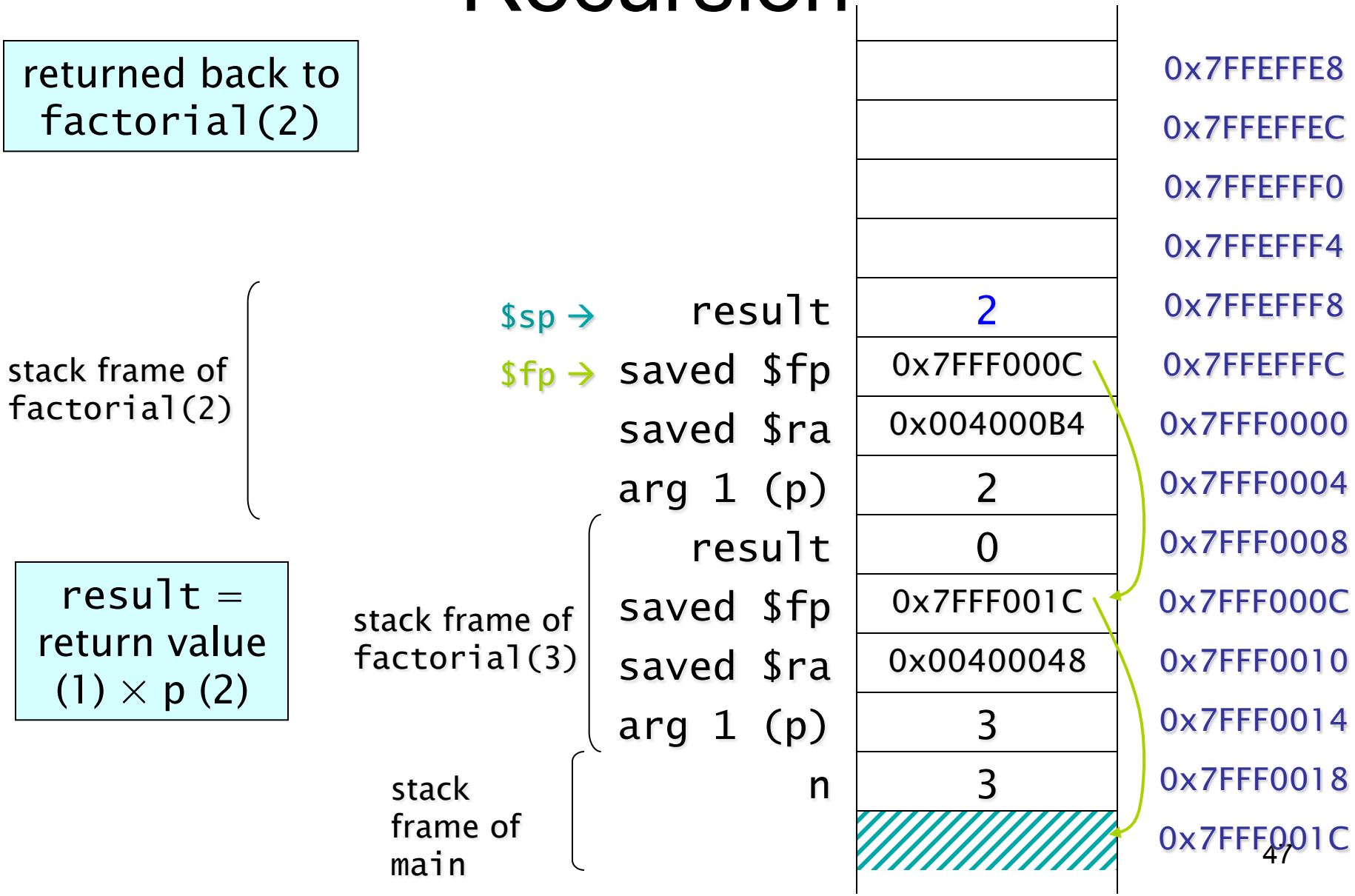


Recursion

$v0 = 1

returned back to factorial(2)

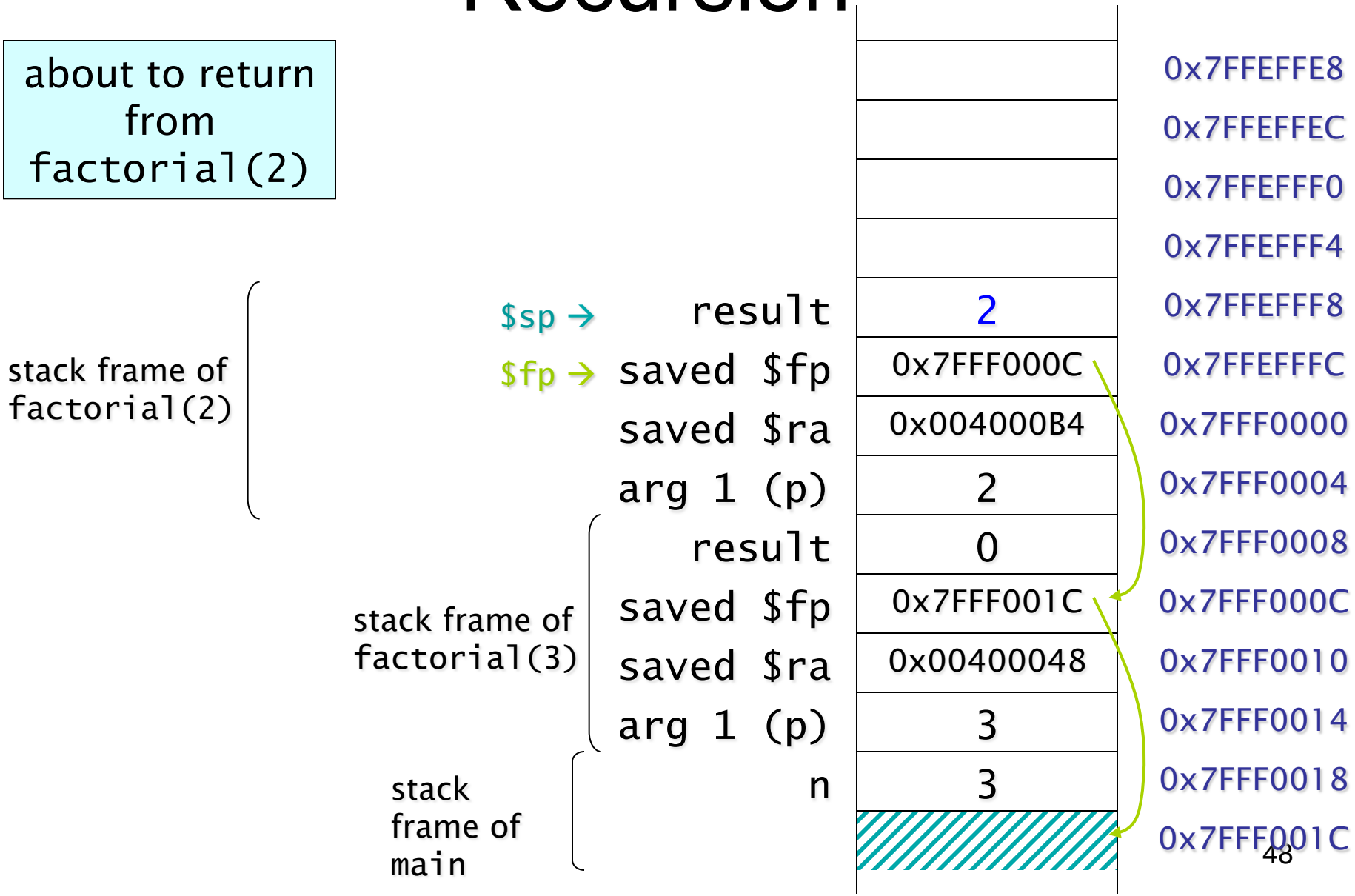
result = return value (1) × p (2)



$v0 = 1

Recursion

about to return from factorial(2)

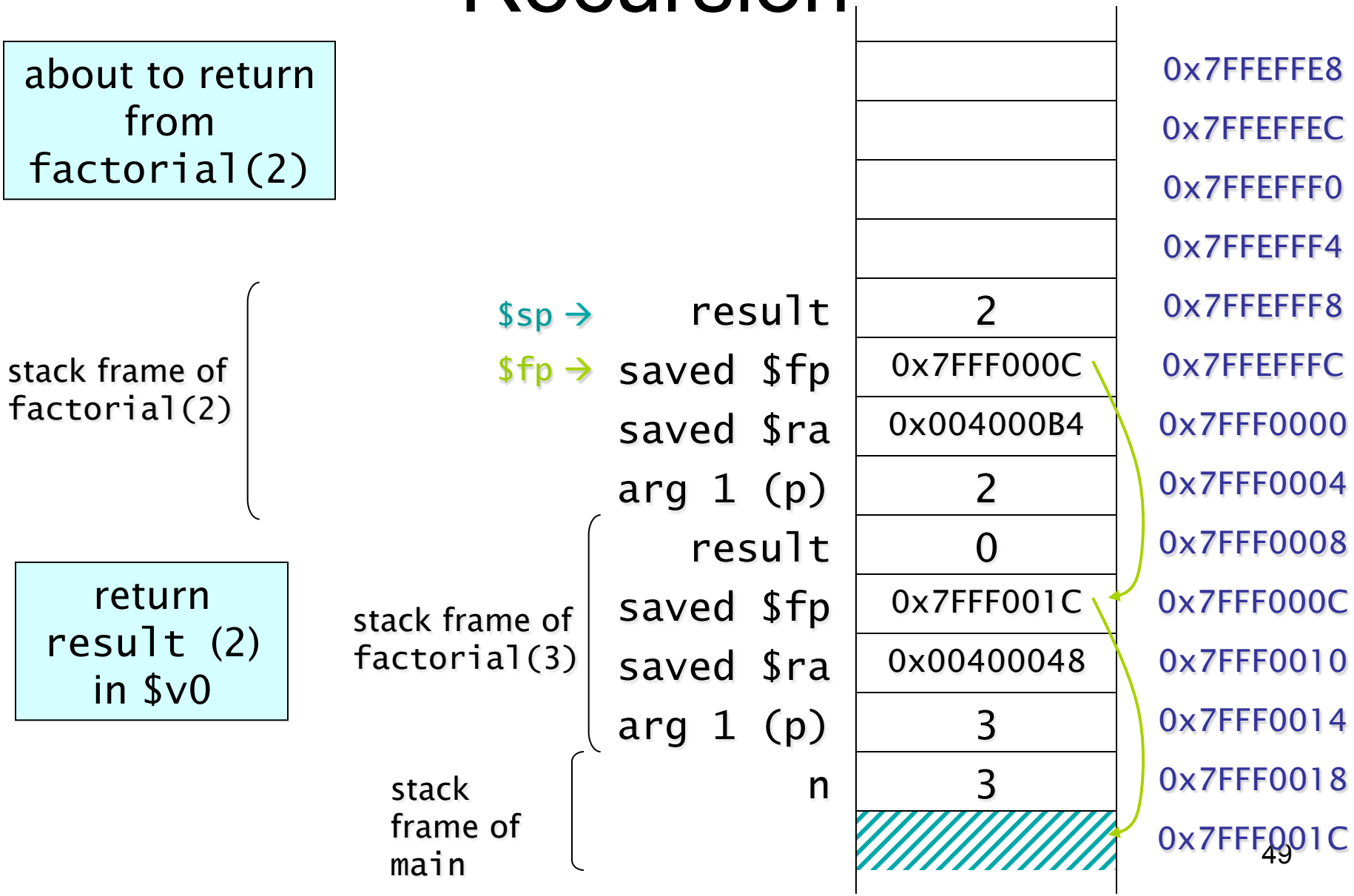


Recursion

$v0 = 2

about to return from factorial(2)

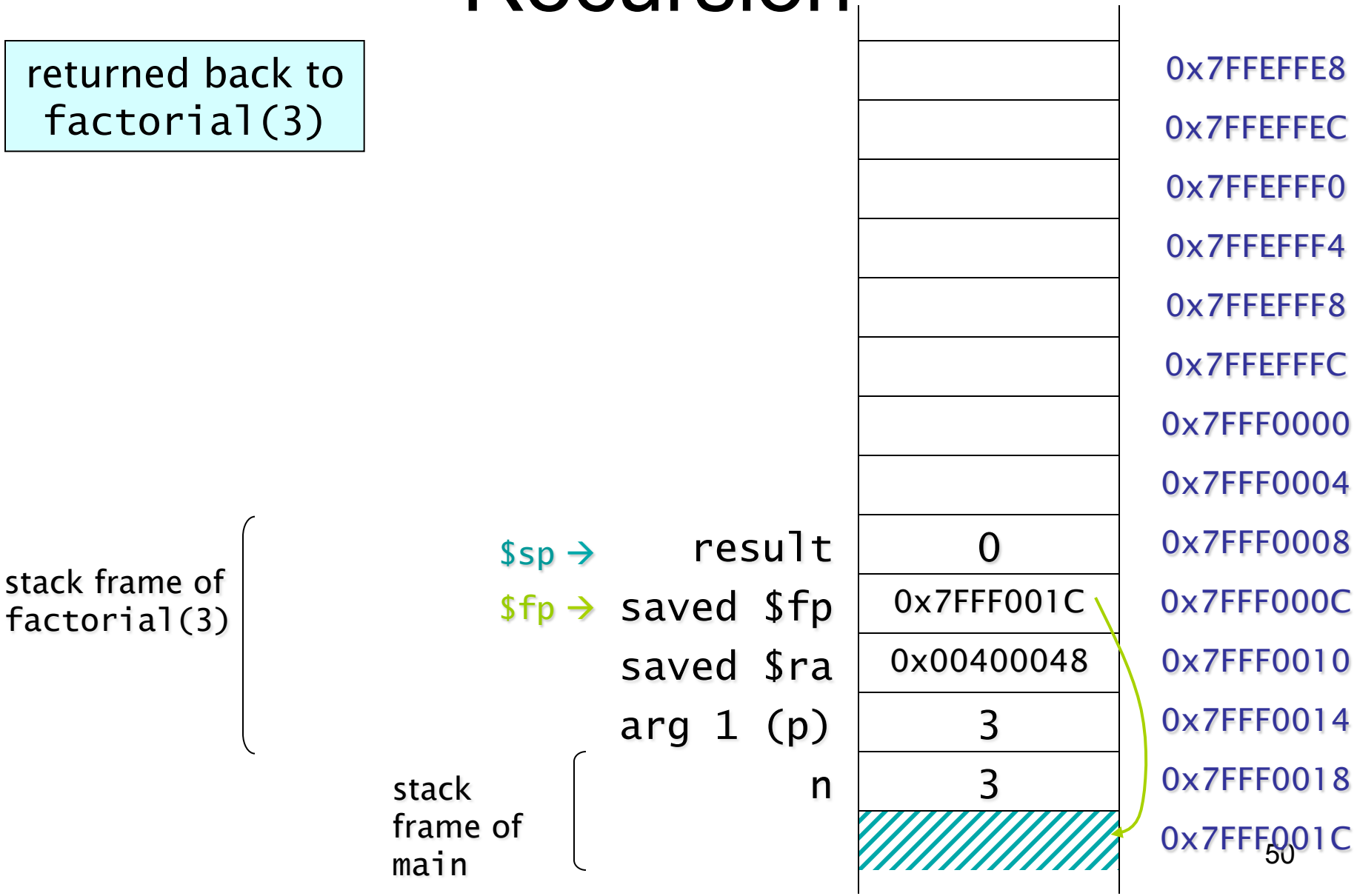
return result (2) in $v0



$v0 = 2

Recursion

returned back to factorial(3)

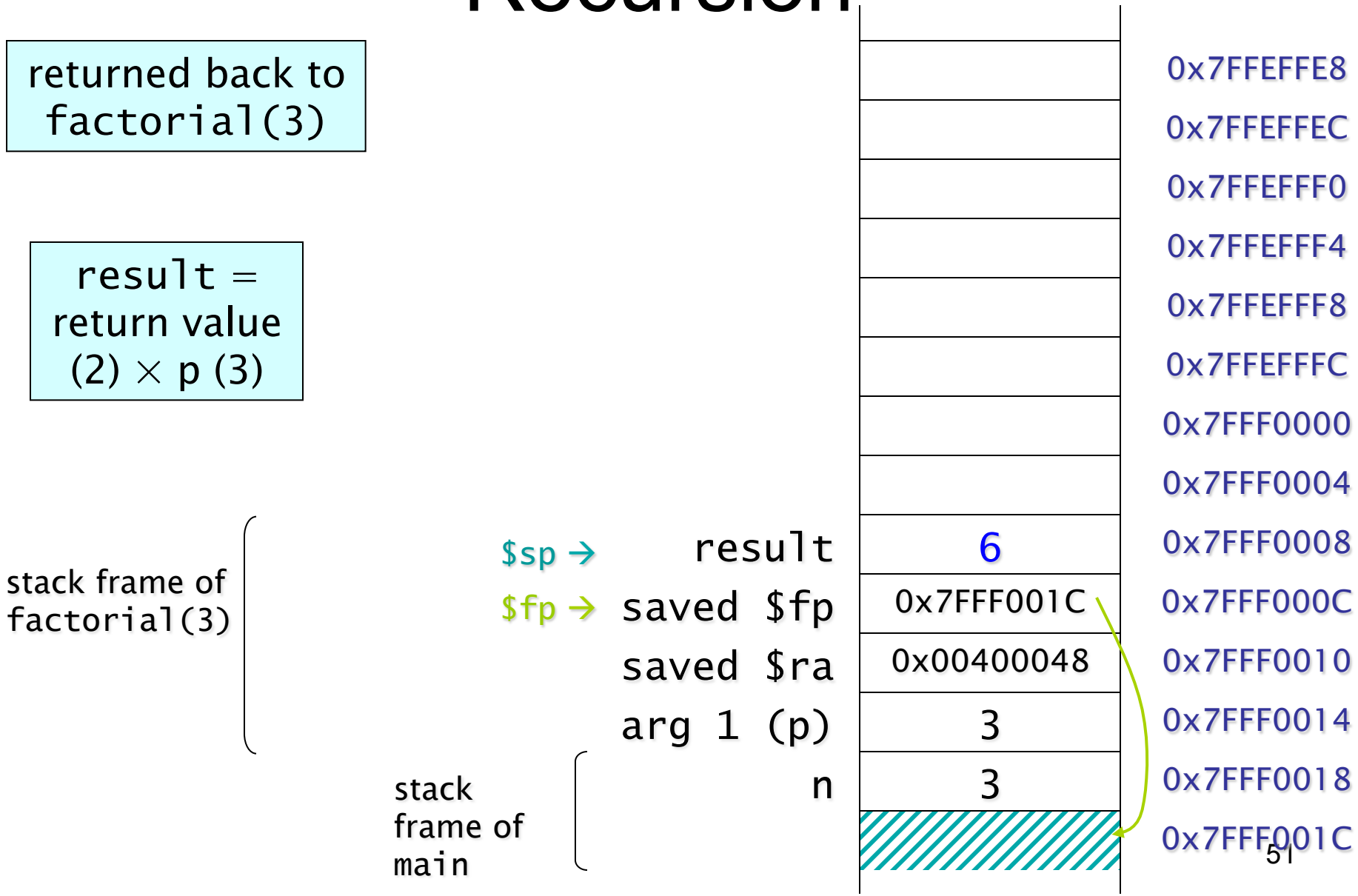


Recursion

$v0 = 2

returned back to factorial(3)

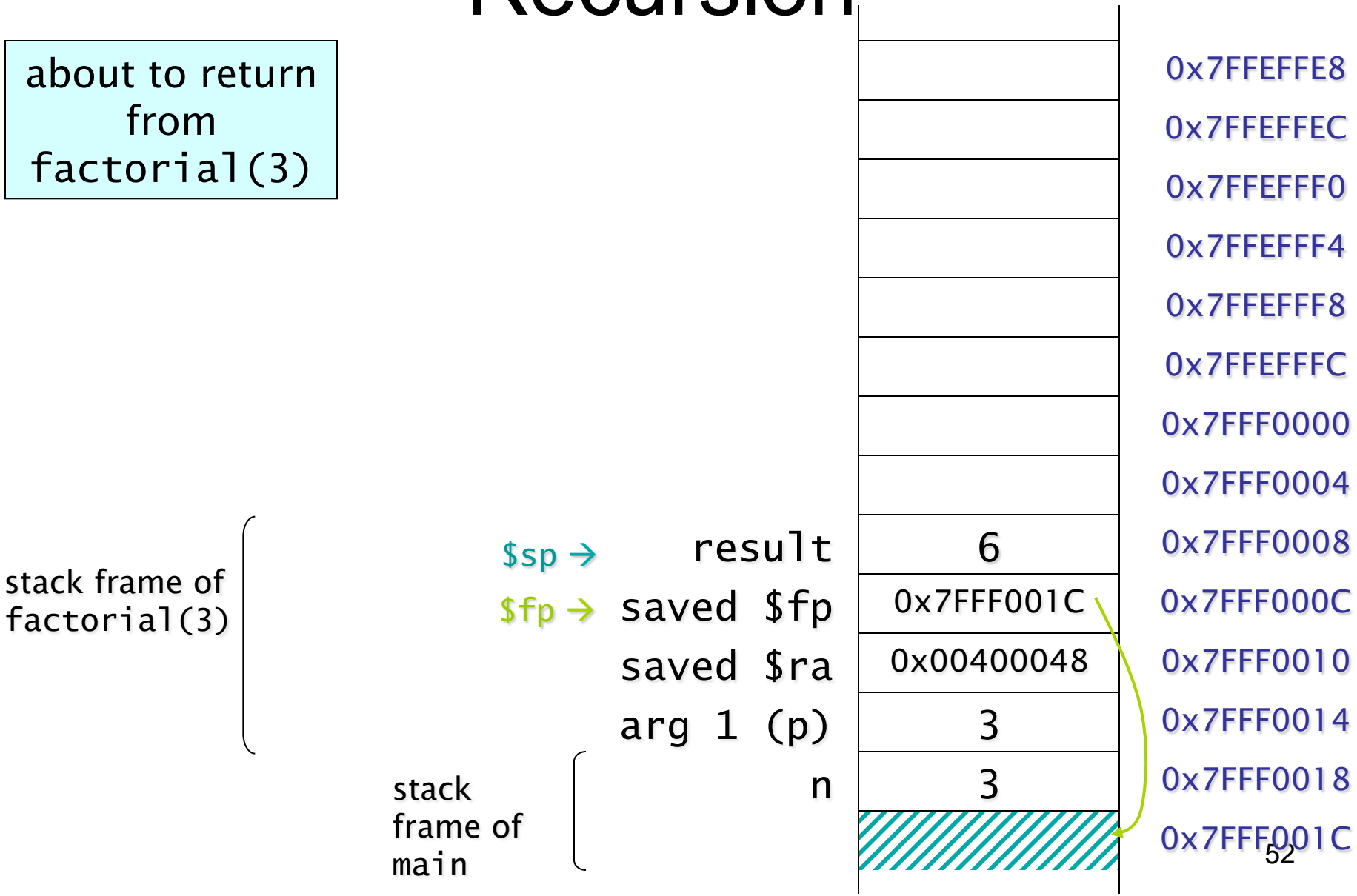
result = return value (2) × p (3)



$v0 = 2

Recursion

about to return from factorial(3)

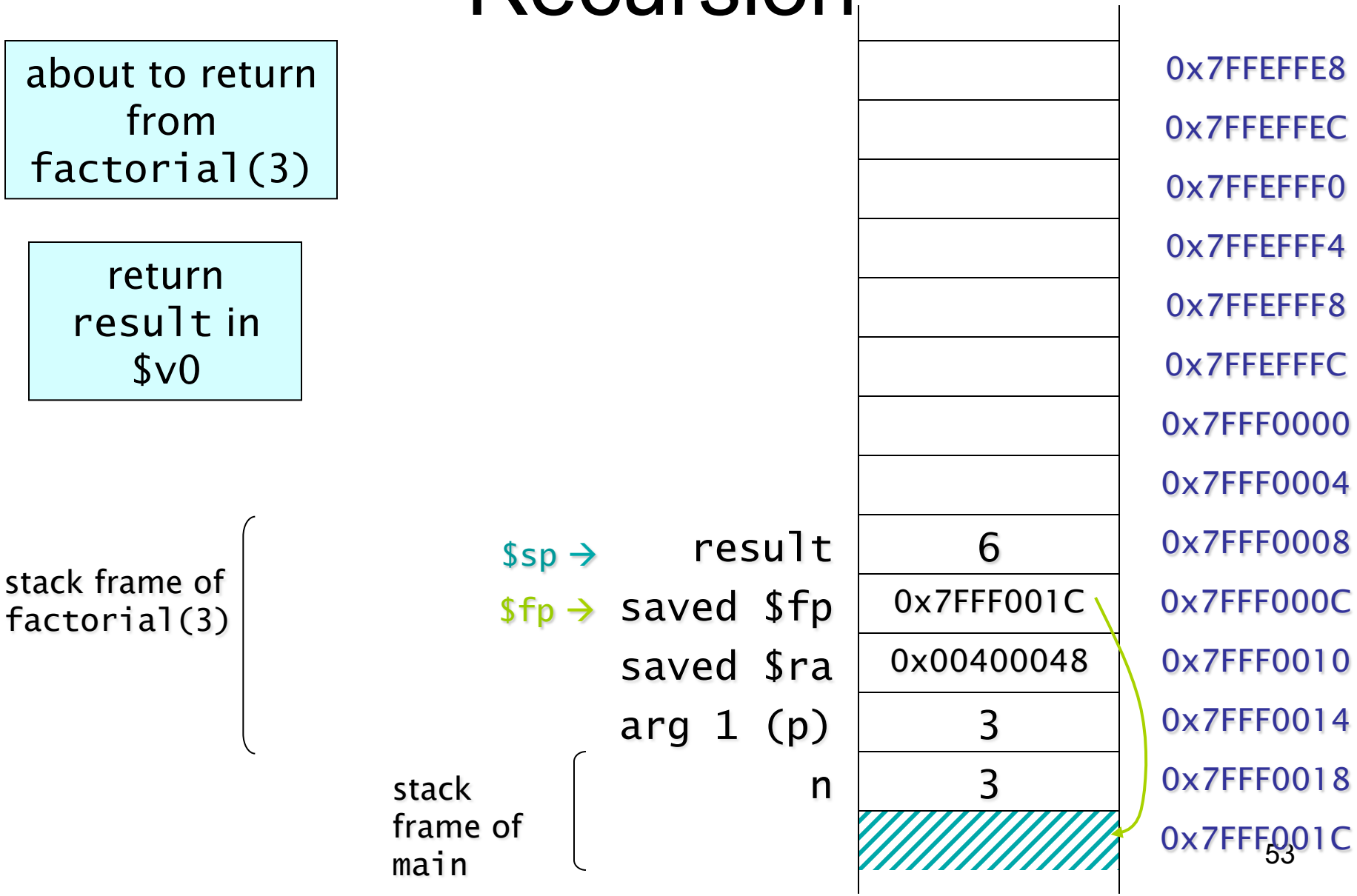


Recursion

$v0 = 6

about to return from factorial(3)

return result in $v0

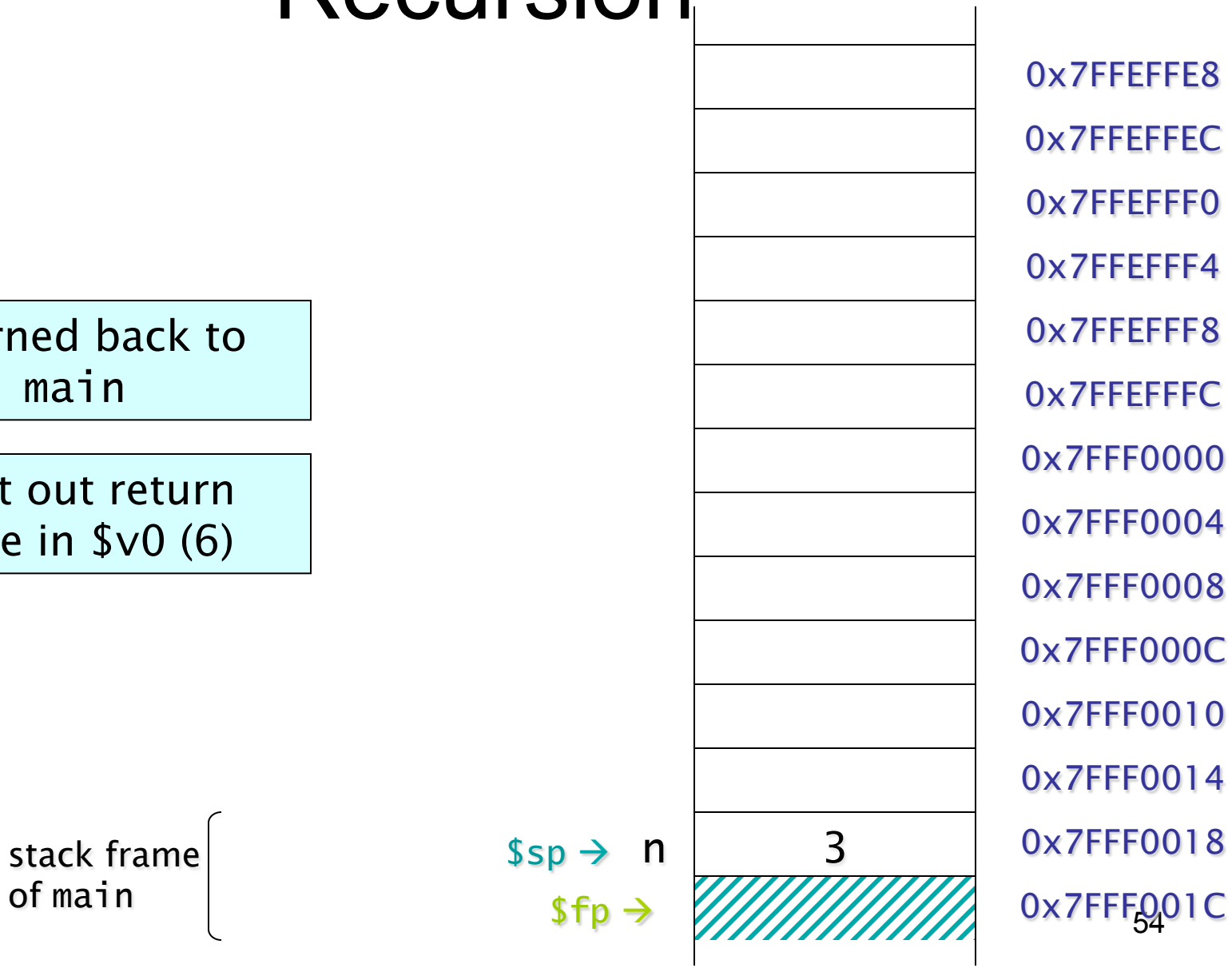


$v0 = ⑥

Recursion

returned back to `main`

print out return value in $v0 (6)



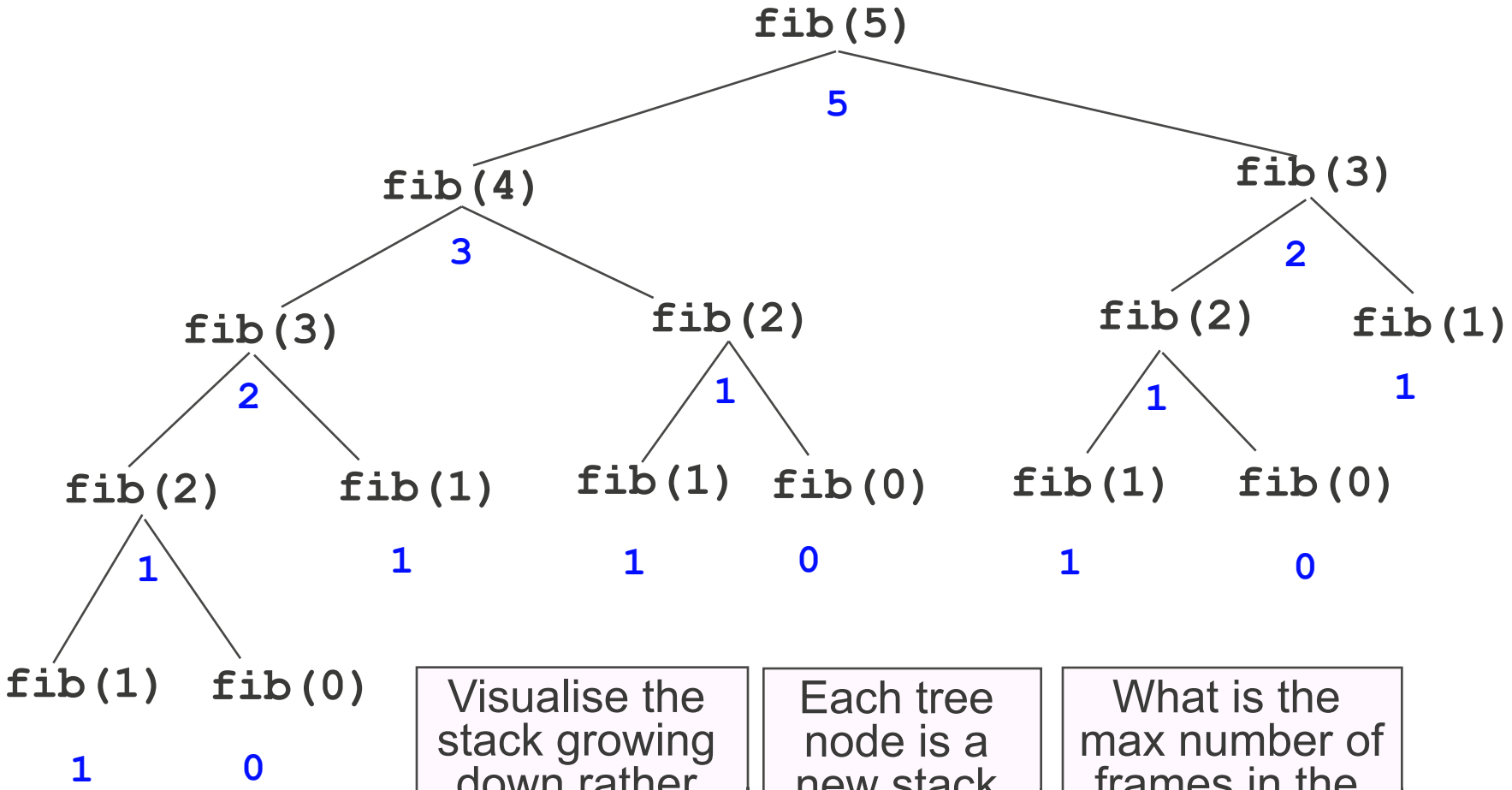
What about non-linear recursion?

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return (fib(n-1) + fib(n-2))
```

What would the evolution of the system stack look like?

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return (fib(n-1) + fib(n-2))
```

fibonacci's execution: call tree



Visualise the stack growing down rather than up

Each tree node is a new stack frame

What is the max number of frames in the stack?

Summary

- Accessing function parameters
- Returning from functions
- Recursion

Going further

- **Official MIPS stack frame convention**
  - Doesn't use `$fp` at all!
  - Slightly more efficient than FIT2085 convention
  - Can be generated by compilers
  - Hard for humans to write/understand